dsPIC30F Programmer's Reference Manual

High Performance
Digital Signal Controllers

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

DNV Certification, Inc.
USA
ANSI ★ RAB
QMS
ACCREDITED
DNV
DNV MSC
The Netherlands
Accredited by the RvA
ISO 9001 / QS-9000
REGISTERED FIRM

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999 and Mountain View, California in March 2002. The Company’s quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, non-volatile memory and analog products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001 certified.

Table of Contents

NOTES:

Section 1. Introduction

HIGHLIGHTS

This section of the manual contains the following topics:

1.1 Introduction

Microchip Technology's focus is on products that meet the needs of the embedded control market. We are a leading supplier of:

- 8-bit general purpose microcontrollers (PICmicro® MCUs)
- dsPIC30F 16-bit microcontrollers
- Speciality and standard non-volatile memory devices
- Security devices (KEELOQ®)
- Application specific standard products

Please request a Microchip Product Line Card for a listing of all the interesting products that we have to offer. This literature can be obtained from your local sales office, or downloaded from the Microchip web site (www.microchip.com).

1.2 Manual Objective

PICmicro and dsPIC30F devices are grouped by the size of their Instruction Word and Data Path. The current device families are:

1. Base-Line: 12-bit Instruction Word length, 8-bit Data Path
2. Mid-Range: 14-bit Instruction Word length, 8-bit Data Path
3. High-End: 16-bit Instruction Word length, 8-bit Data Path
4. Enhanced: 16-bit Instruction Word length, 8-bit Data Path
5. dsPIC30F: 24-bit Instruction Word length, 16-bit Data Path

This manual is a software developer's reference for the dsPIC30F 16-bit MCU family of devices. This manual describes the Instruction Set in detail and also provides general information to assist the user in developing software for the dsPIC30F MCU family.

This manual does not include detailed information about the core, peripherals, system integration or device-specific information. The user should refer to the dsPIC30F MCU Family Reference Manual for information about the core, peripherals and system integration. For device specific information, the user should refer to the data sheet. The information that can be found in the data sheet includes:

- Device memory map
- Device pinout and packaging details
- Device electrical specifications
- List of peripherals included on the device.

Code examples are given throughout this manual. These examples are valid for any device in the dsPIC30F MCU family.

1.3 Development Support

Microchip offers a wide range of development tools that allow users to efficiently develop and debug application code. Microchip's development tools can be broken down into four categories:

1. Code generation
2. Hardware/Software debug
3. Device programmer
4. Product evaluation boards

Information about the latest tools, product briefs and user guides can be obtained from the Microchip web site (www.microchip.com) or from your local Microchip Sales Office.

Microchip offers other reference tools to speed the development cycle. These include:

- Application Notes
- Reference Designs
- Microchip web site
- Local Sales Offices with Field Application Support
- Corporate Support Line

The Microchip web site lists other sites that may be useful references.

1.4 Style and Symbol Conventions

Throughout this document, certain style and font format conventions are used. Most format conventions imply a distinction should be made for the emphasized text. The MCU industry has many symbols and non-conventional word definitions/abbreviations. Table 1-1 provides a description for many of the conventions contained in this document.

Table 1-1: Document Conventions

Symbol or Term	Description
set	To force a bit/register to a value of logic '`1`'.
clear	To force a bit/register to a value of logic '`0`'.
RESET	1) To force a register/bit to its default state. 2) A condition in which the device places itself after a device RESET occurs. Some bits will be forced to '`0`' (such as interrupt enable bits), while others will be forced to '`1`' (such as the I/O data direction bits).
`0xnnnn`	Designates the number 'nnnn' in the hexadecimal number system. These conventions are used in the code examples. For example, `0x013F` or `0xA800`.
: (colon)	Used to specify a range or the concatenation of registers/bits/pins. One example is ACCAU:ACCAH:ACCAL, which is the concatenation of three registers to form the 40-bit accumulator. Concatenation order (left-right) usually specifies a positional relationship (MSb to LSb, higher to lower).
< >	Specifies bit(s) locations in a particular register. One example is SR<IPL2:IPL0> (or IPL<2:0>), which specifies the register and associated bits or bit positions.
MSb, MSbit, LSb, LSbit	Indicates the Least Significant or Most Significant bit in a field.
MSByte, MSWord, LSByte, LSWord	Indicates the Least/Most Significant Byte or Word in a field of bits.
Courier Font	Used for code examples, binary numbers and for Instruction Mnemonics in the text.
Times Font	Used for equations and variables.
Times, Bold Font, Italics	Used in explanatory text for items called out from a graphic/equation/example.
Note:	A Note presents information that we wish to re-emphasize, either to help you avoid a common pitfall, or make you aware of operating differences between some device family members. In most instances, a Note is used in a shaded box (as illustrated below), however when referenced to a table, a Note will stand-alone and immediately follow the associated table (as illustrated below Table 1-2). **Note:** This is a Note in a shaded note box.

1.5 Instruction Set Symbols

The Summary Tables in Section 3-2 and Section 6.5, and the instruction descriptions in Section 5.4 utilize the symbols shown in Table 1-2.

Table 1-2:Symbols Used in Instruction Summary Tables and Descriptions

Symbol	Description
{ }	Optional field or operation
[text]	The location addressed by text
(text)	The contents of text
#text	The literal defined by text
a ∈ [b, c, d]	"a" must be in the set of [b, c, d]
<n:m>	Register bit field
{label:}	Optional label name
Acc	Accumulator A or Accumulator B
AWB	Accumulator Write Back
bit4	4-bit wide bit position (0:7 in Byte mode, 0:15 in Word mode)
Expr	Absolute address, label or expression (resolved by the linker)
f	File register address
lit1	1-bit literal (0:1)
lit4	4-bit literal (0:15)
lit5	5-bit literal (0:31)
lit8	8-bit literal (0:255)
lit10	10-bit literal (0:255 in Byte mode, 0:1023 in Word mode)
lit14	14-bit literal (0:16383)
lit16	16-bit literal (0:65535)
lit23	23-bit literal (0:8388607)
Slit4	Signed 4-bit literal (-8:7)
Slit6	Signed 6-bit literal (-32:31) (range is limited to -16:16)
Slit10	Signed 10-bit literal (-512:511)
Slit16	Signed 16-bit literal (-32768:32767)
TOS	Top-of-Stack
Wb	Base working register
Wd	Destination working register (direct and indirect addressing)
Wm, Wn	Working register divide pair (dividend, divisor)
Wm*Wm	Working register multiplier pair (same source register)
Wm*Wn	Working register multiplier pair (different source registers)
Wn	Both source and destination working register (direct addressing)
Wnd	Destination working register (direct addressing)
Wns	Source working register (direct addressing)
WREG	Default working register (assigned to W0)
Ws	Source working register (direct and indirect addressing)
Wx	Source Addressing mode and working register for X data bus pre-fetch
Wxd	Destination working register for X data bus pre-fetch
Wy	Source Addressing mode and working register for Y data bus pre-fetch
Wyd	Destination working register for Y data bus pre-fetch

Note: The range of each symbol is instruction dependent. Refer to **Section 5. "Instruction Descriptions"** for the specific instruction range.

 Advance Information

1.6 Related Documents

Microchip, as well as other sources, offer additional documentation which can aid in your development with dsPIC30F MCUs. These lists contain the most common documentation, but other documents may also be available. Please check the Microchip web site (www.microchip.com) for the latest published technical documentation.

1.6.1 Microchip Documentation

The following dsPIC30F documentation is available from Microchip at the time of this writing. Many of these documents provide application specific information that gives actual examples of using, programming and designing with dsPIC30F MCUs.

1. **dsPIC30F MCU Family Reference Manual (DS70046)**

   The dsPIC30F Family Reference Manual provides information about the dsPIC30F architecture, peripherals and system integration features. The details of device operation are provided in this document, along with numerous code examples.

2. **dsPIC30F High Performance 16-Bit Digital Signal Controller Family Overview (DS70043)**

   This document provides a summary of the available dsPIC30F family variants, including device pinouts, memory sizes and available peripherals.

3. **dsPIC30F Data Sheets**

   The data sheets contain device specific information, such as pinout and packaging details, electrical specifications, and memory maps. Please check the Microchip web site (www.microchip.com) for a list of available device data sheets.

1.6.2 Third Party Documentation

There are several documents available from third party sources around the world. Microchip does not review these documents for technical accuracy. However, they may be a helpful source for understanding the operation of Microchip dsPIC30F devices. Please refer to the Microchip web site (www.microchip.com) for third party documentation related to the dsPIC30F.

NOTES:

Section 2. Programmer’s Model

HIGHLIGHTS

This section of the manual contains overview information about the dsPIC30F devices. It contains the following major topics:

2.1 dsPIC30F Overview

The dsPIC30F core is a 16-bit (data) modified Harvard architecture with an enhanced instruction set, including support for DSP. The core has a 24-bit instruction word, with a variable length opcode field. The program counter (PC) is 23-bits wide and addresses up to 4M x 24 bits of user program memory space. A single cycle instruction pre-fetch mechanism is used to help maintain throughput and provides predictable execution. The majority of instructions execute in a single cycle, and overhead free program loop constructs are supported using the `DO` and `REPEAT` instructions, both of which are interruptible.

The dsPIC30F has sixteen, 16-bit working registers. Each of the working registers can act as a data, address or offset register. The 16th working register (W15) operates as a software stack pointer for interrupts and calls.

The dsPIC30F instruction set has two classes of instructions: the MCU class of instructions and the DSP class of instructions. These two instruction classes are seamlessly integrated into the architecture and execute from a single execution unit. The instruction set includes many Addressing modes and was designed for optimum C compiler efficiency.

The data space can be addressed as 32K words or 64 Kbytes and is split into two blocks, referred to as X and Y data memory. Each memory block has its own independent Address Generation Unit (AGU). The MCU class of instructions operate solely through the X memory AGU, which accesses the entire memory map as one linear data space. The DSP dual source class of instructions operates through the X and Y AGUs, which splits the data address space into two parts. The X and Y data space boundary is arbitrary and device specific.

The upper 32 Kbytes of the data space memory map can optionally be mapped into program space at any 16K program word boundary, defined by the 8-bit Program Space Visibility Page (PSVPAG) register. The program to data space mapping feature lets any instruction access program space as if it were data space, which is useful for storing data coefficients.

Overhead free circular buffers (modulo addressing) are supported in both X and Y address spaces. The modulo addressing removes the software boundary checking overhead for DSP algorithms. Furthermore, the X AGU circular addressing can be used with any of the MCU class of instructions. The X AGU also supports bit-reverse addressing, to greatly simplify input or output data reordering for radix-2 FFT algorithms.

The core supports Inherent (no operand), Relative, Literal, Memory Direct, Register Direct, Register Indirect and Register Offset Addressing modes. Each instruction is associated with a predefined Addressing mode group, depending upon its functional requirements. As many as 7 Addressing modes are supported for each instruction.

For most instructions, the dsPIC30F is capable of executing a data (or program data) memory read, a working register (data) read, a data memory write and a program (instruction) memory read per instruction cycle. As a result, 3-operand instructions can be supported, allowing A+B=C operations to be executed in a single cycle.

The DSP engine features a high speed, 17-bit by 17-bit multiplier, a 40-bit ALU, two 40-bit saturating accumulators and a 40-bit bi-directional barrel shifter. The barrel shifter is capable of shifting a 40-bit value, up to 16-bits right, or up to 16-bits left, in a single cycle. The DSP instructions operate seamlessly with all other instructions and have been designed for optimal real-time performance. The `MAC` instruction and other associated instructions can concurrently fetch two data operands from memory while multiplying two working registers. This requires that the data space be split for these instructions and linear for all others. This is achieved in a transparent and flexible manner through dedicating certain working registers to each address space.

The dsPIC30F has a vectored exception scheme with up to 8 sources of non-maskable traps and 54 interrupt sources. Each interrupt source can be assigned to one of seven priority levels.

2.2 Programmer's Model

The programmer's model diagram for the dsPIC30F is shown in Figure 2-1.

All registers in the programmer's model are memory mapped and can be manipulated directly by the instruction set. A description of each register is provided in Table 2-1.

Table 2-1: Programmer's Model Register Descriptions

Register	Description
ACCA, ACCB	40-bit DSP Accumulators
CORCON	CPU Core Configuration register
DCOUNT	DO Loop Count register
DOEND	DO Loop End Address register
DOSTART	DO Loop Start Address register
PC	23-bit Program Counter
PSVPAG	Program Space Visibility Page Address register
RCOUNT	Repeat Loop Count register
SPLIM	Stack Pointer Limit Value register
SR	ALU and DSP Engine Status register
TBLPAG	Table Memory Page Address register
W0 - W15	Working register array

2.2.1 Working Register Array

The 16 working (W) registers can function as data, address or offset registers. The function of a W register is determined by the instruction that accesses it.

Byte instructions, which target the working register array, only affect the Least Significant Byte of the target register. Since the working registers are memory mapped, the Least *and* Most Significant Bytes can be manipulated through byte wide data memory space accesses.

2.2.2 Default Working Register (WREG)

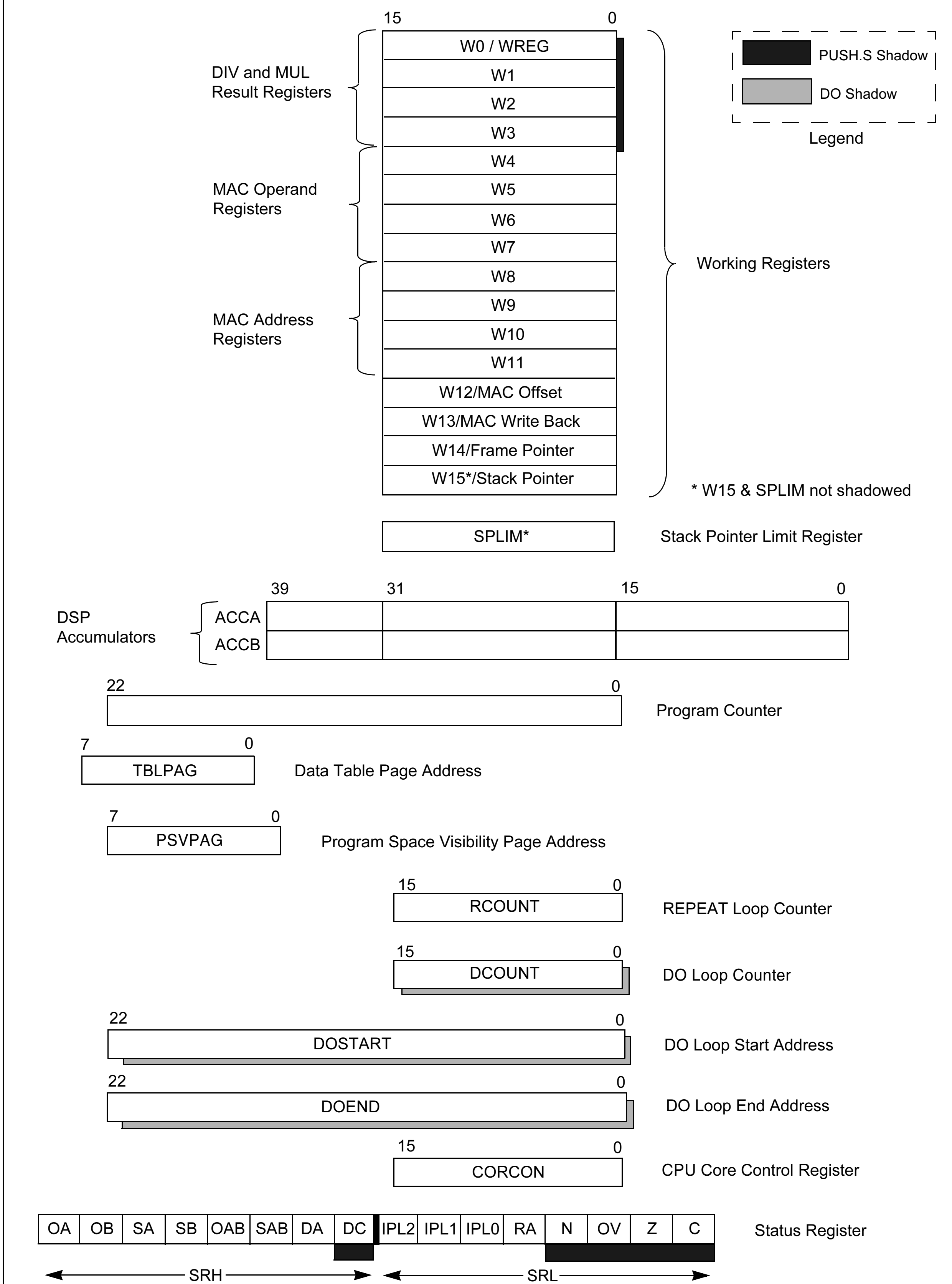
The dsPIC30F instruction set can be divided into two instruction types: working register instructions and file register instructions. The working register instructions use the working register array as data values, or as addresses that point to a memory location. In contrast, file register instructions operate on a specific memory address contained in the instruction opcode.

File register instructions that also utilize a working register do not specify the working register that is to be used for the instruction. Instead, a default working register (WREG) is used for these file register instructions. Working register W0 is assigned to be the WREG. The WREG assignment is not programmable.

2.2.3 Software Stack Frame Pointer

A frame is a user defined section of memory in the stack, used by a function to allocate memory for local variables. W14 has been assigned for use as a stack frame pointer with the link (`LNK`) and unlink (`ULNK`) instructions. However, if a stack frame pointer and the `LNK` and `ULNK` instructions are not used, W14 can be used by any instruction in the same manner as all other W registers. See **Section 4.7.3 "Software Stack Frame Pointer"** for detailed information about the Frame Pointer.

Figure 2-1: Programmer's Model Diagram



2.2.4 Software Stack Pointer

W15 serves as a dedicated software stack pointer, and will be automatically modified by function calls, exception processing and returns. However, W15 can be referenced by any instruction in the same manner as all other W registers. This simplifies reading, writing and manipulating the stack pointer. Refer to **Section 4.7.1 "Software Stack Pointer"** for detailed information about the stack pointer.

2.2.5 Stack Pointer Limit Register (SPLIM)

The SPLIM is a 16-bit register associated with the stack pointer. It is used to prevent the stack pointer from overflowing and accessing memory beyond the user allocated region of stack memory. Refer to **Section 4.7.5 "Stack Pointer Overflow"** for detailed information about the SPLIM.

2.2.6 Accumulator A, Accumulator B

Accumulator A (ACCA) and Accumulator B (ACCB) are 40-bit wide registers, utilized by DSP instructions to perform mathematical and shifting operations. Each accumulator is composed of 3 memory mapped registers:

- AccxU (bits 39 - 32)
- AccxH (bits 31 - 16)
- AccxL (bits 15 - 0)

Refer to **Section 4.12 "Accumulator Usage"** for details on using ACCA and ACCB.

2.2.7 Program Counter

The Program Counter (PC) is 23-bits wide. Instructions are addressed in the 4M x 24-bit user program memory space by PC<22:1>, where PC<0> is always set to '0' to maintain instruction word alignment and provide compatibility with data space addressing. This means that during normal instruction execution, the PC increments by 2.

Program memory located at 0x80000000 and above is utilized for device configuration data, Unit ID and Device ID. This region is not available for user code execution and the PC can not access this area. However, one may access this region of memory using Table instructions. Refer to the dsPIC30F MCU Family Reference Manual for details on accessing the configuration data, Unit ID and Device ID.

2.2.8 TBLPAG Register

The TBLPAG register is used to hold the upper 8 bits of a program memory address during table read and write operations. Table instructions are used to transfer data between program memory space and data memory space. Refer to the dsPIC30F MCU Family Reference Manual for details on accessing program memory with the Table instructions.

2.2.9 PSVPAG Register

Program space visibility allows the user to map a 32 Kbyte section of the program memory space into the upper 32 Kbytes of data address space. This feature allows transparent access of constant data through dsPIC30F instructions that operate on data memory. The PSVPAG register selects the 32 Kbyte region of program memory space that is mapped to the data address space. Refer to the dsPIC30F MCU Family Reference Manual for details on program space visibility.

2.2.10 RCOUNT Register

The 14-bit RCOUNT register contains the loop counter for the `REPEAT` instruction. When a `REPEAT` instruction is executed, RCOUNT is loaded with the repeat count of the instruction, either "lit14" for the "`REPEAT #lit14`" instruction, or the contents of Wn for the "`REPEAT Wn`" instruction. The REPEAT loop will be executed RCOUNT+1 times.

> **Note 1:** If a REPEAT loop is executing and gets interrupted, RCOUNT may be cleared by the Interrupt Service Routine to break out of the REPEAT loop when the foreground code is re-entered.
>
> **2:** Refer to the dsPIC30F Family Reference Manual for complete details about REPEAT loops.

2.2.11 DCOUNT Register

The 14-bit DCOUNT register contains the loop counter for hardware DO loops. When a `DO` instruction is executed, DCOUNT is loaded with the loop count of the instruction, either "lit14" for the "`DO #lit14,Expr`" instruction, or the 14 Least Significant bits of Ws for the "`DO Ws,Expr`" instruction. The DO loop will be executed DCOUNT+1 times.

> **Note 1:** DCOUNT contains a shadow register. See **Section 2.2.16 "Shadow Registers"** for information on shadowing.
>
> **2:** Refer to the dsPIC30F Family Reference Manual for complete details about DO loops.

2.2.12 DOSTART Register

The DOSTART register contains the starting address for a hardware DO loop. When a `DO` instruction is executed, DOSTART is loaded with the address of the instruction following the `DO` instruction. This location in memory is the start of the DO loop. When looping is activated, program execution continues with the instruction stored at the DOSTART address after the last instruction in the DO loop is executed. This mechanism allows for zero overhead looping.

> **Note 1:** DOSTART has a shadow register. See **Section 2.2.16 "Shadow Registers"** for information on shadowing.
>
> **2:** Refer to the dsPIC30F Family Reference Manual for complete details about DO loops.

2.2.13 DOEND Register

The DOEND register contains the ending address for a hardware DO loop. When a `DO` instruction is executed, DOEND is loaded with the address specified by the expression in the `DO` instruction. This location in memory specifies the last instruction in the DO loop. When looping is activated and the instruction stored at the DOEND address is executed, program execution will continue from the DO loop start address (stored in the DOSTART register).

> **Note 1:** DOEND has a shadow register. See **Section 2.2.16 "Shadow Registers"** for information on shadowing.
>
> **2:** Refer to the dsPIC30F Family Reference Manual for complete details about DO loops.

2.2.14 Status Register

The 16-bit Status register, shown in Register 2-1, maintains status information for instructions which have most recently been executed. Operation status bits exist for MCU operations, loop operations and DSP operations. Additionally, the Status register contains the CPU Interrupt Priority Level bits, IPL<2:0>, which are used for interrupt processing.

2.2.14.1 MCU ALU Status Bits

The MCU operation status bits are either affected or used by the majority of instructions in the instruction set. Most of the Logic, Math, Rotate/Shift and Bit instructions modify the MCU status bits after execution, and the conditional Branch instructions use the state of individual status bits to determine the flow of program execution. All conditional Branch instructions are listed in **Section 4.8 "Conditional Branch Instructions"**.

The Carry, Zero, Overflow, Negative and Digit Carry (C, Z, OV, N and DC) bits are used to show the immediate status of the MCU ALU. They indicate when an operation has resulted in a carry, zero, overflow, negative result and digit carry, respectively. When a subtract operation is performed, the C flag is used as a Borrow flag.

The Z status bit is a special zero status bit that is useful for extended precision arithmetic. The Z bit functions like a normal Z flag for all instructions except those that use a carry or borrow input (`ADDC, CPB, SUBB` and `SUBBR`). See **Section 4.9 "Z Status Bit"** for usage of the Z status bit.

Note 1:	All MCU bits are shadowed during execution of the `PUSH.S` instruction and they are restored on execution of the `POP.S` instruction.
2:	All MCU bits, except the DC flag (which is not in the SRL), are stacked during exception processing (see **Section 4.7.1 "Software Stack Pointer"**).

2.2.14.2 Loop Status Bits

The DO Active and REPEAT Active (DA, RA) bits are used to indicate when looping is active. The `DO` instructions affect the DA flag, which indicates that a DO loop is active. The DA flag is set to '`1`' when the first instruction of the DO loop is executed, and it is cleared when the last instruction of the loop completes final execution. Likewise, the RA flag indicates that a `REPEAT` instruction is being executed, and it is only affected by the `REPEAT` instructions. The RA flag is set to '`1`' when the instruction being repeated begins execution, and it is cleared when the instruction being repeated completes execution for the last time.

The DA flag is read only. This means that looping may not be initiated by writing a '`1`' to DA, nor may looping be terminated by writing a '`0`' to DA. If a DO loop must be terminated prematurely, the EDT bit, CORCON<11>, should be used.

Since the RA flag is also read only, it may not be directly cleared. However, if a `REPEAT` or its target instruction is interrupted, the Interrupt Service Routine may clear the RA flag of the SRL, which resides on the stack. This action will disable looping once program execution returns from the Interrupt Service Routine, because the restored RA will be '`0`'.

2.2.14.3 DSP ALU Status Bits

The high byte of the Status Register (SRH) is used by the DSP class of instructions, and it is modified when data passes through one of the adders. The SRH provides status information about overflow and saturation for both accumulators. The Saturate A, Saturate B, Overflow A and Overflow B (SA, SB, OA, OB) bits provide individual accumulator status, while the Saturate AB and Overflow AB (SAB, OAB) bits provide combined accumulator status. The SAB and OAB bits provide the software developer efficiency in checking the register for saturation or overflow.

The OA and OB bits are used to indicate when an operation has generated an overflow into the guard bits (bits 32 through 39) of the respective accumulator. This condition can only occur when the processor is in Super Saturation mode, or if saturation is disabled. It indicates that the operation has generated a number which cannot be represented with the lower 31 bits of the accumulator.

The SA and SB bits are used to indicate when an operation has generated an overflow out of the Most Significant bit of the respective accumulator. The SA and SB bits are active, regardless of the Saturation mode (Disabled, Normal or Super) and may be considered “sticky”. Namely, once the SA or SB is set to ‘`1`’, it can only be cleared manually by software, regardless of subsequent DSP operations. When required, it is recommended that the bits be cleared with the `BCLR` instruction.

For convenience, the OA and OB bits are logically ORed together to form the OAB flag, and the SA and SB bits are logically ORed to form the SAB flag. These cumulative status bits provide efficient overflow and saturation checking when an algorithm is implemented, which utilizes both accumulators. Instead of interrogating the OA and the OB bits independently for arithmetic overflows, a single check of OAB may be performed. Likewise, when checking for saturation, SAB may be examined instead of checking both the SA and SB bits. Note that clearing the SAB flag will clear both the SA and SB bits.

2.2.14.4 Interrupt Priority Level Status Bits

The three IPL bits of the SRL, SR<7:5>, and the IPL3 bit, CORCON<3>, set the CPU’s Interrupt Priority Level (IPL) which is used for exception processing. Exceptions consist of interrupts and hardware traps. Interrupts have a user defined priority level between 0 and 7, while traps have a fixed priority level between 8 and 15. The fourth Interrupt Priority Level bit, IPL3, is a special IPL bit that may only be read or cleared by the user. This bit is only set when a hardware trap is activated and it is cleared after the trap is serviced.

The CPU’s IPL identifies the lowest level exception which may interrupt the processor. The interrupt level of a pending exception must always be greater than the CPU’s IPL for the CPU to process the exception. This means that if the IPL is ‘0’, all exceptions at priority Level 1 and above may interrupt the processor. If the IPL is ‘7’, only hardware traps may interrupt the processor.

When an exception is serviced, the IPL is automatically set to the priority level of the exception being serviced, which will disable all exceptions of equal and lower priority. However, since the IPL field is read/write, one may modify the lower three bits of the IPL in an Interrupt ServiceRoutine to control which exceptions may preempt the exception processing. Since the SRL is stacked during exception processing, the original IPL is always restored after the exception is serviced. If required, one may also prevent exceptions from nesting by setting the NSTDIS bit, INTCON1<15>.

Note:	Refer to the dsPIC30F Family Reference Manual for complete details on exception processing.

2.2.15 Core Control Register

The 16-bit CPU Core Control Register (CORCON), shown in Register 2-2, is used to set the configuration of the dsPIC30F CPU. This register provides the ability to:

- map program space into data space
- set the ACCA and ACCB saturation enable
- set the Data Space Write Saturation mode
- set the Accumulator Saturation and Rounding modes
- set the Multiplier mode for DSP operations
- terminate DO loops prematurely

On device RESET, the CORCON is set to `0x0020`, which sets the following mode:

- Program Space not Mapped to Data Space (**PSV** = `0`)
- ACCA and ACCB Saturation Disabled (**SATA** = 0, **SATB** = `0`)
- Data Space Write Saturation Enabled (**SATDW** = `1`)
- Accumulator Saturation mode set to normal (**ACCSAT** = `0`)
- Accumulator Rounding mode set to unbiased (**RND** = `0`)
- DSP Multiplier mode set to signed fractional (**US** = `0`, **IF** = `0`)

In addition to setting CPU modes, the CORCON contains status information about the DO loop nesting level (**DL**<2:0>) and the **IPL**<3> status bit, which indicates if a trap exception is being processed.

2.2.16 Shadow Registers

A shadow register is used as a temporary holding register and can transfer its contents to or from the associated host register upon some event. Some of the registers in the programmer's model have a shadow register, which is utilized during the execution of a `DO, POP.S` or `PUSH.S` instruction. Shadow register usage is shown in Table 2-2.

Table 2-2: Automatic Shadow Register Usage

Location	DO	POP.S/PUSH.S
DCOUNT	Yes	—
DOSTART	Yes	—
DOEND	Yes	—
Status Register - DC, N, OV, Z and C bits	—	Yes
W0 - W3	—	Yes

Since the DCOUNT, DOSTART and DOEND registers are shadowed, the ability to nest DO loops without additional overhead is provided. Since all shadow registers are one register deep, up to one level of DO loop nesting is possible. Further nesting of DO loops is possible in software, with support provided by the DO Loop Nesting Level Status bits in the CORCON, CORCON<10:8>.

> **Note:** All shadow registers are one register deep and are not directly accessible. Additional shadowing may be performed in software using the software stack.

Register 2-1: SR, Status Register

High Byte (SRH):							
R-0	R-0	R/C-0	R/C-0	R-0	R/C-0	R-0	R/W-0
OA	OB	SA	SB	OAB	SAB	DA	DC
bit 15							bit 8

Low Byte (SRL):							
R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
IPL<2:0>			RA	N	OV	Z	C
bit 7							bit 0

bit 15 **OA:** Accumulator A Overflow bit
1 = Accumulator A overflowed
0 = Accumulator A has not overflowed

bit 14 **OB:** Accumulator B Overflow bit
1 = Accumulator B overflowed
0 = Accumulator B has not overflowed

bit 13 **SA:** Accumulator A Saturation bit
1 = Accumulator A is saturated or has been saturated at some time
0 = Accumulator A is not saturated

Note 1: This bit may be read or cleared, but not set.
2: Once this bit is set, it must be cleared manually by software.

bit 12 **SB:** Accumulator B Saturation bit
1 = Accumulator B is saturated or has been saturated at some time
0 = Accumulator B is not saturated

Note 1: This bit may be read or cleared, but not set.
2: Once this bit is set, it must be cleared manually by software.

bit 11 **OAB:** OA || OB Combined Accumulator Overflow bit
1 = Accumulators A or B have overflowed
0 = Neither Accumulators A or B have overflowed

bit 10 **SAB:** SA || SB Combined Accumulator bit
1 = Accumulators A or B are saturated or have been saturated at some time in the past
0 = Neither Accumulators A or B are saturated

Note 1: This bit may be read or cleared, but not set.
2: Once this bit is set, it must be cleared manually by software.
3: Clearing this bit will clear SA and SB.

bit 9 **DA:** DO Loop Active bit
1 = DO loop in progress
0 = DO loop not in progress

Note: This bit is read only.

bit 8 **DC:** MCU ALU Half Carry bit
1 = A carry-out from the Most Significant bit of the lower nibble occurred
0 = No carry-out from the Most Significant bit of the lower nibble occurred

bit 7-5 **IPL<2:0>:** Interrupt Priority Level bits
111 = CPU Interrupt Priority Level is 7 (15). User interrupts disabled.
110 = CPU Interrupt Priority Level is 6 (14)
101 = CPU Interrupt Priority Level is 5 (13)
100 = CPU Interrupt Priority Level is 4 (12)
011 = CPU Interrupt Priority Level is 3 (11)
010 = CPU Interrupt Priority Level is 2 (10)
001 = CPU Interrupt Priority Level is 1 (9)
000 = CPU Interrupt Priority Level is 0 (8)

Note: The IPL<2:0> bits are concatenated with the IPL<3> bit (CORCON<3>) to form the CPU Interrupt Priority Level. The value in parentheses indicates the IPL, if IPL<3> = 1.

Register 2-1: SR, Status Register (Continued)

bit 4 **RA:** REPEAT Loop Active bit
`1` = REPEAT loop in progress
`0` = REPEAT loop not in progress

bit 3 **N:** MCU ALU Negative bit
`1` = The result of the operation was negative
`0` = The result of the operation was not negative

bit 2 **OV:** MCU ALU Overflow bit
`1` = Overflow occurred
`0` = No overflow occurred

bit 1 **Z:** MCU ALU Zero bit
`1` = The result of the operation was zero
`0` = The result of the operation was not zero

Note: Refer to **Section 4.9 "Z Status Bit"** for operation with `ADDC, CPB, SUBB` and `SUBBR` instructions.

bit 0 **C:** MCU ALU Carry/$\overline{\text{Borrow}}$ bit
`1` = A carry-out from the Most Significant bit occurred
`0` = No carry-out from the Most Significant bit occurred

Legend:		
R = Readable bit	W = Writable bit	C = Clearable bit
-n = Value at POR	1 = bit is set	0 = bit is cleared

Register 2-2: CORCON, Core Control Register

High Byte:							
U	U	U	R/W-0	R(0)/W-0	R-0	R-0	R/W-0
—	—	—	US	EDT	DL<2:0>		
bit 15							bit 8

Low Byte:							
R/W-0	R/W-0	R/W-1	R/W-0	R/C-0	R/W-0	R/W-0	R/W-0
SATA	SATB	SATDW	ACCSAT	IPL3	PSV	RND	IF
bit 7							bit 0

bit 15-13 Unused

bit 12 **US:** Unsigned or Signed Multiplier Mode Select bit
`1` = Unsigned mode enabled for DSP multiply operations
`0` = Signed mode enabled for DSP multiply operations

bit 11 **EDT:** Early DO Loop Termination Control bit
`1` = Terminate executing DO loop at end of current iteration
`0` = No effect

Note: This bit will always read ‘0’.

bit 10-8 **DL<2:0>:** DO Loop Nesting Level Status bits
`111` = DO looping is nested at 7 levels
`110` = DO looping is nested at 6 levels
`110` = DO looping is nested at 5 levels
`110` = DO looping is nested at 4 levels
`011` = DO looping is nested at 3 levels
`010` = DO looping is nested at 2 levels
`001` = DO looping is active, but not nested (just 1 level)
`000` = DO looping is not active

Note 1: DL<2:1> are read only.
2: The first two levels of DO loop nesting are handled by hardware.

bit 7 **SATA:** ACCA Saturation Enable bit
`1` = Accumulator A saturation enabled
`0` = Accumulator A saturation disabled

bit 6 **SATB:** ACCB Saturation Enable bit
`1` = Accumulator B saturation enabled
`0` = Accumulator B saturation disabled

bit 5 **SATDW:** Data Space Write from DSP Engine Saturation Enable bit
`1` = Data space write saturation enabled
`0` = Data space write saturation disabled

bit 4 **ACCSAT:** Accumulator Saturation Mode Select bit
`1` = 9.31 saturation (Super Saturation)
`0` = 1.31 saturation (Normal Saturation)

bit 3 **IPL3:** Interrupt Priority Level 3 Status bit
`1` = CPU Interrupt Priority Level is 8 or greater (trap exception activated)
`0` = CPU Interrupt Priority Level is 7 or less (no trap exception activated)

Note 1: This bit may be read or cleared, but not set.
2: This bit is concatenated with the IPL<2:0> bits (SR<7:5>) to form the CPU Interrupt Priority Level.

bit 2 **PSV:** Program Space Visibility in Data Space Enable bit
`1` = Program space visible in data space
`0` = Program space not visible in data space

Register 2-2: CORCON, Core Control Register (Continued)

bit 1 **RND:** Rounding Mode Select bit
1 = Biased (conventional) rounding enabled
0 = Unbiased (convergent) rounding enabled

bit 0 **IF:** Integer or Fractional Multiplier Mode Select bit
1 = Integer mode enabled for DSP multiply operations
0 = Fractional mode enabled for DSP multiply operations

Legend:			
R = Readable bit	W = Writable bit	C = Clearable bit	x = bit is unknown
-n = Value at POR	1 = bit is set	0 = bit is cleared	U = Unimplemented bit, read as '0'

NOTES:

Section 3. Instruction Set Overview

HIGHLIGHTS

This section of the manual contains the following major topics:

3.1 Introduction

The dsPIC30F instruction set provides a broad suite of instructions, which supports traditional microcontroller applications and a class of instructions, which supports math intensive applications. Since almost all of the functionality of the PICmicro® MCU instruction set has been maintained, this hybrid instruction set allows a friendly DSP migration path for users already familiar with the PICmicro microcontroller.

3.2 Instruction Set Overview

The dsPIC30F instruction set contains 84 instructions, which can be grouped into the ten functional categories shown in Table 3-1. Table 1-2 defines the symbols used in the instruction summary tables, Table 3-2 through Table 3-11. These tables define the syntax, description, storage and execution requirements for each instruction. Storage requirements are represented in 24-bit instruction words and execution requirements are represented in instruction cycles.

Table 3-1: dsPIC30F Instruction Groups

Functional Group	Summary Table	Page #
Move Instructions	Table 3-2	3-3
Math Instructions	Table 3-3	3-4
Logic Instructions	Table 3-4	3-5
Rotate/Shift Instructions	Table 3-5	3-6
Bit Instructions	Table 3-6	3-7
Compare/Skip Instructions	Table 3-7	3-8
Program Flow Instructions	Table 3-8	3-9
Shadow/Stack Instructions	Table 3-9	3-10
Control Instructions	Table 3-10	3-10
DSP Instructions	Table 3-11	3-10

Most instructions have several different Addressing modes and execution flows, which require different instruction variants. For instance, there are six unique `ADD` instructions and each instruction variant has its own instruction encoding. Instruction format descriptions and specific instruction operation are provided in **Section 3. "Instruction Set Overview"**. Additionally, a composite alphabetized instruction set table is provided in **Section 6. "Reference"**.

3.2.1 Multi-Cycle Instructions

As the instruction summary tables show, most instructions execute in a single cycle, with the following exceptions:

- Instructions `DO, MOV.D, POP.D, PUSH.D, TBLRDH, TBLRDL, TBLWTH and TBLWTL` require 2 cycles to execute.
- Instructions `DIV.S, DIV.U` and `DIVF` are single cycle instructions, which should be executed 18 consecutive times as the target of a `REPEAT` instruction.
- Instructions that change the program counter also require 2 cycles to execute, with the extra cycle executed as a `NOP`. `SKIP` instructions, which skip over a 2-word instruction, require 3 instruction cycles to execute, with 2 cycles executed as a `NOP`.
- The `RETFIE, RETLW and RETURN` are a special case of an instruction that changes the program counter. These execute in 3 cycles, unless an exception is pending and then they execute in 2 cycles.

Note:	Instructions which access program memory as data, using Program Space Visibility, will incur a one or two cycle delay. However, when the target instruction of a REPEAT loop accesses program memory as data, only the first execution of the target instruction is subject to the delay. See the dsPIC30F Family Reference Manual for details.

3.2.2 Multi-Word Instructions

As defined by **Subsection Table 3-2: "Move Instructions"** , almost all instructions consume one instruction word (24-bits), with the exception of the `CALL, DO` and `GOTO` instructions, which are Program Flow Instructions, listed in Table 3-8. These instructions require two words of memory because their opcodes embed large literal operands.

3.3 Instruction Set Summary Tables

Table 3-2: Move Instructions

Assembly Syntax		Description	Words	Cycles	Page #
EXCH	Wns,Wnd	Swap Wns and Wnd	1	1	5-115
MOV	f {,WREG}(see Note)	Move f to destination	1	1	5-145
MOV	WREG,f	Move WREG to f	1	1	5-146
MOV	f,Wnd	Move f to Wnd	1	1	5-147
MOV	Wns,f	Move Wns to f	1	1	5-148
MOV.B	#lit8,Wnd	Move 8-bit literal to Wnd	1	1	5-149
MOV	#lit16,Wnd	Move 16-bit literal to Wnd	1	1	5-150
MOV	[Ws+Slit10],Wnd	Move [Ws + signed 10-bit offset] to Wnd	1	1	5-151
MOV	Wns,[Wd+Slit10]	Move Wns to [Wd + signed 10-bit offset]	1	1	5-152
MOV	Ws,Wd	Move Ws to Wd	1	1	5-153
MOV.D	Ws,Wnd	Move double Ws to Wnd:Wnd+1	1	2	5-155
MOV.D	Wns,Wd	Move double Wns:Wns+1 to Wd	1	2	5-157
SWAP	Wn	Wn = byte or nibble swap Wn	1	1	5-249
TBLRDH	Ws,Wd	Read high program word to Wd	1	2	5-250
TBLRDL	Ws,Wd	Read low program word to Wd	1	2	5-252
TBLWTH	Ws,Wd	Write Ws to high program word	1	2	5-254
TBLWTL	Ws,Wd	Write Ws to low program word	1	2	5-256

Note: When the optional {,WREG} operand is specified, the destination of the instruction is WREG. When {,WREG} is not specified, the destination of the instruction is the file register f.

Table 3-3: Math Instructions

Assembly Syntax		Description	Words	Cycles	Page #
ADD	f {,WREG}(1)	Destination = f + WREG	1	1	5-7
ADD	#lit10,Wn	Wn = lit10 + Wn	1	1	5-8
ADD	Wb,#lit5,Wd	Wd = Wb + lit5	1	1	5-9
ADD	Wb,Ws,Wd	Wd = Wb + Ws	1	1	5-10
ADDC	f {,WREG}(1)	Destination = f + WREG + (C)	1	1	5-14
ADDC	#lit10,Wn	Wn = lit10 + Wn + (C)	1	1	5-15
ADDC	Wb,#lit5,Wd	Wd = Wb + lit5 + (C)	1	1	5-16
ADDC	Wb,Ws,Wd	Wd = Wb + Ws + (C)	1	1	5-17
DAW.B	Wn	Wn = decimal adjust Wn	1	1	5-95
DEC	f {,WREG}(1)	Destination = f – 1	1	1	5-96
DEC	Ws,Wd	Wd = Ws – 1	1	1	5-97
DEC2	f {,WREG}(1)	Destination = f – 2	1	1	5-98
DEC2	Ws,Wd	Wd = Ws – 2	1	1	5-99
DIV.S	Wm, Wn	Signed 16/16-bit integer divide	1	18(2)	5-101
DIV.SD	Wm, Wn	Signed 32/16-bit integer divide	1	18(2)	5-101
DIV.U	Wm, Wn	Unsigned 16/16-bit integer divide	1	18(2)	5-103
DIV.UD	Wm, Wn	Unsigned 32/16-bit integer divide	1	18(2)	5-103
DIVF	Wm, Wn	Signed 16/16-bit fractional divide	1	18(2)	5-105
INC	f {,WREG}(1)	Destination = f + 1	1	1	5-124
INC	Ws,Wd	Wd = Ws + 1	1	1	5-125
INC2	f {,WREG}(1)	Destination = f + 2	1	1	5-126
INC2	Ws,Wd	Wd = Ws + 2	1	1	5-127
MUL	f	W3:W2 = f * WREG	1	1	5-169
MUL.SS	Wb,Ws,Wnd	{Wnd+1,Wnd} = sign(Wb) * sign(Ws)	1	1	5-170
MUL.SU	Wb,#lit5,Wnd	{Wnd+1,Wnd} = sign(Wb) * unsign(lit5)	1	1	5-172
MUL.SU	Wb,Ws,Wnd	{Wnd+1,Wnd} = sign(Wb) * unsign(Ws)	1	1	5-174
MUL.US	Wb,Ws,Wnd	{Wnd+1,Wnd} = unsign(Wb) * sign(Ws)	1	1	5-176
MUL.UU	Wb,#lit5,Wnd	{Wnd+1,Wnd} = unsign(Wb) * unsign(lit5)	1	1	5-178
MUL.UU	Wb,Ws,Wnd	{Wnd+1,Wnd} = unsign(Wb) * unsign(Ws)	1	1	5-179
SE	Ws,Wnd	Wnd = sign-extended Ws	1	1	5-220
SUB	f {,WREG}(1)	Destination = f – WREG	1	1	5-230
SUB	#lit10,Wn	Wn = Wn – lit10	1	1	5-231
SUB	Wb,#lit5,Wd	Wd = Wb – lit5	1	1	5-232
SUB	Wb,Ws,Wd	Wd = Wb – Ws	1	1	5-233
SUBB	f {,WREG}(1)	Destination = f – WREG – ($\overline{C}$)	1	1	5-236
SUBB	#lit10,Wn	Wn = Wn – lit10 – ($\overline{C}$)	1	1	5-237
SUBB	Wb,#lit5,Wd	Wd = Wb – lit5 – ($\overline{C}$)	1	1	5-238
SUBB	Wb,Ws,Wd	Wd = Wb – Ws – ($\overline{C}$)	1	1	5-239
SUBBR	f {,WREG}(1)	Destination = WREG – f – ($\overline{C}$)	1	1	5-241
SUBBR	Wb,#lit5,Wd	Wd = lit5 – Wb – ($\overline{C}$)	1	1	5-242
SUBBR	Wb,Ws,Wd	Wd = Ws – Wb – ($\overline{C}$)	1	1	5-243
SUBR	f {,WREG}(1)	Destination = WREG – f	1	1	5-245
SUBR	Wb,#lit5,Wd	Wd = lit5 – Wb	1	1	5-246
SUBR	Wb,Ws,Wd	Wd = Ws – Wb	1	1	5-247
ZE	Ws,Wnd	Wnd = zero-extended Ws	1	1	5-264

Note 1: When the optional {,WREG} operand is specified, the destination of the instruction is WREG. When {,WREG} is not specified, the destination of the instruction is the file register f.

2: The divide instructions must be preceded with a "`REPEAT #17`" instruction, such that they are executed 18 consecutive times.

Table 3-4: Logic Instructions

Assembly Syntax		Description	Words	Cycles	Page #
AND	f {,WREG}(see Note)	Destination = f .AND. WREG	1	1	5-19
AND	#lit10,Wn	Wn = lit10 .AND. Wn	1	1	5-20
AND	Wb,#lit5,Wd	Wd = Wb .AND. lit5	1	1	5-21
AND	Wb,Ws,Wd	Wd = Wb .AND. Ws	1	1	5-22
CLR	f	f = `0x0000`	1	1	5-75
CLR	WREG	WREG = `0x0000`	1	1	5-75
CLR	Wd	Wd = `0x0000`	1	1	5-76
COM	f {,WREG}(see Note)	Destination = $\overline{f}$	1	1	5-80
COM	Ws,Wd	Wd = $\overline{Ws}$	1	1	5-81
IOR	f {,WREG}(see Note)	Destination = f .IOR. WREG	1	1	5-128
IOR	#lit10,Wn	Wn = lit10 .IOR. Wn	1	1	5-129
IOR	Wb,#lit5,Wd	Wd = Wb .IOR. lit5	1	1	5-130
IOR	Wb,Ws,Wd	Wd = Wb .IOR. Ws	1	1	5-131
NEG	f {,WREG}(see Note)	Destination = $\overline{f}$ + `1`	1	1	5-181
NEG	Ws,Wd	Wd = $\overline{Ws}$ + `1`	1	1	5-182
SETM	f	f = `0xFFFF`	1	1	5-221
SETM	WREG	WREG = `0xFFFF`	1	1	5-221
SETM	Wd	Wd = `0xFFFF`	1	1	5-222
XOR	f {,WREG}(see Note)	Destination = f .XOR. WREG	1	1	5-259
XOR	#lit10,Wn	Wn = lit10 .XOR. Wn	1	1	5-260
XOR	Wb,#lit5,Wd	Wd = Wb .XOR. lit5	1	1	5-261
XOR	Wb,Ws,Wd	Wd = Wb .XOR. Ws	1	1	5-262

Note: When the optional {,WREG} operand is specified, the destination of the instruction is WREG. When {,WREG} is not specified, the destination of the instruction is the file register f.

Table 3-5: Rotate/Shift Instructions

Assembly Syntax		Description	Words	Cycles	Page #
ASR	f {,WREG}(see Note)	Destination = arithmetic right shift f	1	1	5-24
ASR	Ws,Wd	Wd = arithmetic right shift Ws	1	1	5-25
ASR	Wb,#lit4,Wnd	Wnd = arithmetic right shift Wb by lit4	1	1	5-27
ASR	Wb,Wns,Wnd	Wnd = arithmetic right shift Wb by Wns	1	1	5-28
LSR	f {,WREG}(see Note)	Destination = logical right shift f	1	1	5-136
LSR	Ws,Wd	Wd = logical right shift Ws	1	1	5-137
LSR	Wb,#lit4,Wnd	Wnd = logical right shift Wb by lit4	1	1	5-139
LSR	Wb,Wns,Wnd	Wnd = logical right shift Wb by Wns	1	1	5-140
RLC	f {,WREG}(see Note)	Destination = rotate left through Carry f	1	1	5-204
RLC	Ws,Wd	Wd = rotate left through Carry Ws	1	1	5-205
RLNC	f {,WREG}(see Note)	Destination = rotate left (no Carry) f	1	1	5-207
RLNC	Ws,Wd	Wd = rotate left (no Carry) Ws	1	1	5-208
RRC	f {,WREG}(see Note)	Destination = rotate right through Carry f	1	1	5-210
RRC	Ws,Wd	Wd = rotate right through Carry Ws	1	1	5-211
RRNC	f {,WREG}(see Note)	Destination = rotate right (no Carry) f	1	1	5-213
RRNC	Ws,Wd	Wd = rotate right (no Carry) Ws	1	1	5-214
SL	f {,WREG}(see Note)	Destination = left shift f	1	1	5-225
SL	Ws,Wd	Wd = left shift Ws	1	1	5-226
SL	Wb,#lit4,Wnd	Wnd = left shift Wb by lit4	1	1	5-228
SL	Wb,Wns,Wnd	Wnd = left shift Wb by Wns	1	1	5-229

Note: When the optional {,WREG} operand is specified, the destination of the instruction is WREG. When {,WREG} is not specified, the destination of the instruction is the file register f.

Table 3-6: Bit Instructions

Assembly Syntax		Description	Words	Cycles	Page #
BCLR	f,#bit4	Bit clear f	1	1	5-29
BCLR	Ws,#bit4	Bit clear Ws	1	1	5-30
BSET	f,#bit4	Bit set f	1	1	5-54
BSET	Ws,#bit4	Bit set Ws	1	1	5-55
BSW.C	Ws,Wb	Write C bit to Ws<Wb>	1	1	5-56
BSW.Z	Ws,Wb	Write $\overline{Z}$ bit to Ws<Wb>	1	1	5-56
BTG	f,#bit4	Bit toggle f	1	1	5-58
BTG	Ws,#bit4	Bit toggle Ws	1	1	5-59
BTST	f,#bit4	Bit test f	1	1	5-67
BTST.C	Ws,#bit4	Bit test Ws to C	1	1	5-68
BTST.Z	Ws,#bit4	Bit test Ws to Z	1	1	5-68
BTST.C	Ws,Wb	Bit test Ws<Wb> to C	1	1	5-69
BTST.Z	Ws,Wb	Bit test Ws<Wb> to Z	1	1	5-69
BTSTS	f,#bit4	Bit test f then set f	1	1	5-71
BTSTS.C	Ws,#bit4	Bit test Ws to C then set Ws	1	1	5-72
BTSTS.Z	Ws,#bit4	Bit test Ws to Z then set Ws	1	1	5-72
FBCL	Ws,Wnd	Find bit change from left (MSb) side	1	1	5-116
FF1L	Ws,Wnd	Find first one from left (MSb) side	1	1	5-118
FF1R	Ws,Wnd	Find first one from right (LSb) side	1	1	5-120

Table 3-7: Compare/Skip Instructions

Assembly Syntax		Description	Words	Cycles(see Note)	Page #
BTSC	f,#bit4	Bit test f, skip if clear	1	1 (2 or 3)	5-60
BTSC	Ws,#bit4	Bit test Ws, skip if clear	1	1 (2 or 3)	5-62
BTSS	f,#bit4	Bit test f, skip if set	1	1 (2 or 3)	5-64
BTSS	Ws,#bit4	Bit test Ws, skip if set	1	1 (2 or 3)	5-65
CP	f	Compare (f – WREG)	1	1	5-82
CP	Wb,#lit5	Compare (Wb – lit5)	1	1	5-83
CP	Wb,Ws	Compare (Wb – Ws)	1	1	5-84
CP0	f	Compare (f – `0x0000`)	1	1	5-85
CP0	Ws	Compare (Ws – `0x0000`)	1	1	5-86
CPB	f	Compare with Borrow (f – WREG – $\overline{C}$)	1	1	5-87
CPB	Wb,#lit5	Compare with Borrow (Wb – lit5 – $\overline{C}$)	1	1	5-88
CPB	Wb,Ws	Compare with Borrow (Wb – Ws – $\overline{C}$)	1	1	5-89
CPSEQ	Wb, Wn	Compare (Wb – Wn), skip if =	1	1 (2 or 3)	5-91
CPSGT	Wb, Wn	Compare (Wb – Wn), skip if >	1	1 (2 or 3)	5-92
CPSLT	Wb, Wn	Compare (Wb – Wn), skip if <	1	1 (2 or 3)	5-93
CPSNE	Wb, Wn	Compare (Wb – Wn), skip if ≠	1	1 (2 or 3)	5-94

Note: Conditional skip instructions execute in 1 cycle if the skip is not taken, 2 cycles if the skip is taken over a one-word instruction and 3 cycles if the skip is taken over a two-word instruction.

Table 3-8: Program Flow Instructions

Assembly Syntax		Description	Words	Cycles	Page #
BRA	Expr	Branch unconditionally	1	2	5-31
BRA	Wn	Computed branch	1	2	5-32
BRA	C,Expr	Branch if Carry (no Borrow)	1	1 (2)(1)	5-33
BRA	GE,Expr	Branch if greater than or equal	1	1 (2)(1)	5-35
BRA	GEU,Expr	Branch if unsigned greater than or equal	1	1 (2)(1)	5-33
BRA	GT,Expr	Branch if greater than	1	1 (2)(1)	5-37
BRA	GTU,Expr	Branch if unsigned greater than	1	1 (2)(1)	5-38
BRA	LE,Expr	Branch if less than or equal	1	1 (2)(1)	5-39
BRA	LEU,Expr	Branch if unsigned less than or equal	1	1 (2)(1)	5-40
BRA	LT,Expr	Branch if less than	1	1 (2)(1)	5-41
BRA	LTU,Expr	Branch if unsigned less than	1	1 (2)(1)	5-44
BRA	N,Expr	Branch if Negative	1	1 (2)(1)	5-43
BRA	NC,Expr	Branch if not Carry (Borrow)	1	1 (2)(1)	5-44
BRA	NN,Expr	Branch if not Negative	1	1 (2)(1)	5-45
BRA	NOV,Expr	Branch if not Overflow	1	1 (2)(1)	5-46
BRA	NZ,Expr	Branch if not Zero	1	1 (2)(1)	5-47
BRA	OA,Expr	Branch if Accumulator A Overflow	1	1 (2)(1)	5-48
BRA	OB,Expr	Branch if Accumulator B Overflow	1	1 (2)(1)	5-49
BRA	OV,Expr	Branch if Overflow	1	1 (2)(1)	5-50
BRA	SA,Expr	Branch if Accumulator A Saturate	1	1 (2)(1)	5-51
BRA	SB,Expr	Branch if Accumulator B Saturate	1	1 (2)(1)	5-52
BRA	Z,Expr	Branch if Zero	1	1 (2)(1)	5-53
CALL	Expr	Call subroutine	2	2	5-73
CALL	Wn	Call indirect subroutine	1	2	5-74
DO	#lit14,Expr	Do code through PC+Expr, (lit14+1) times	2	2	5-107
DO	Wn,Expr	Do code through PC+Expr, (Wn+1) times	2	2	5-109
GOTO	Expr	Go to address	2	2	5-122
GOTO	Wn	Go to address indirectly	1	2	5-123
RCALL	Expr	Relative call	1	2	5-196
RCALL	Wn	Computed call	1	2	5-196
REPEAT	#lit14	Repeat next instruction (lit14+1) times	1	1	5-197
REPEAT	Wn	Repeat next instruction (Wn+1) times	1	1	5-198
RETFIE		Return from interrupt enable	1	3 (2)(2)	5-201
RETLW	#lit10,Wn	Return with lit10 in Wn	1	3 (2)(2)	5-202
RETURN		Return from subroutine	1	3 (2)(2)	5-203

Note 1: Conditional branch instructions execute in 1 cycle if the branch is not taken, or 2 cycles if the branch is taken.

2: `RETURN` instructions execute in 3 cycles, but if an exception is pending, they execute in 2 cycles.

3

Instruction Set Overview

Table 3-9: Shadow/Stack Instructions

Assembly Syntax		Description	Words	Cycles	Page #
LNK	#lit14	Link frame pointer	1	1	5-135
POP	f	Pop TOS to f	1	1	5-186
POP	Wd	Pop TOS to Wd	1	1	5-187
POP.D	Wnd	Double pop from TOS to Wnd:Wnd+1	1	2	5-188
POP.S		Pop shadow registers	1	1	5-189
PUSH	f	Push f to TOS	1	1	5-190
PUSH	Ws	Push Ws to TOS	1	1	5-191
PUSH.D	Wns	Push double Wns:Wns+1 to TOS	1	2	5-192
PUSH.S		Push shadow registers	1	1	5-193
ULNK		Unlink frame pointer	1	1	5-258

Table 3-10: Control Instructions

Assembly Syntax		Description	Words	Cycles	Page #
CLRWDT		Clear Watchdog Timer	1	1	5-79
DISI	#lit14	Disable interrupts for (lit14+1) instruction cycles	1	1	5-100
NOP		No operation	1	1	5-184
NOPR		No operation	1	1	5-185
PWRSAV	#lit1	Enter Power Saving mode lit1	1	1	5-194
RESET		Software device RESET	1	1	5-200

Table 3-11: DSP Instructions

Assembly Syntax		Description	Words	Cycles	Page #
ADD	Acc	Add accumulators	1	1	5-11
ADD	Ws,#Slit4,Acc	16-bit signed add to Acc	1	1	5-12
CLR	Acc,Wx,Wxd,Wy,Wyd,AWB	Clear Acc	1	1	5-77
ED	Wm*Wm,Acc,Wx,Wy,Wxd	Euclidean distance (no accumulate)	1	1	5-111
EDAC	Wm*Wm,Acc,Wx,Wy,Wxd	Euclidean distance	1	1	5-113
LAC	Ws,#Slit4,Acc	Load Acc	1	1	5-133
MAC	Wm*Wn,Acc,Wx,Wxd,Wy, Wyd,AWB	Multiply and accumulate	1	1	5-141
MAC	Wm*Wm,Acc,Wx,Wxd,Wy,Wyd	Square and accumulate	1	1	5-143
MOVSAC	Acc,Wx,Wxd,Wy,Wyd,AWB	Move Wx to Wxd and Wy to Wyd	1	1	5-159
MPY	Wm*Wn,Acc,Wx,Wxd,Wy,Wyd	Multiply Wn by Wm to Acc	1	1	5-161
MPY	Wm*Wm,Acc,Wx,Wxd,Wy,Wyd	Square to Acc	1	1	5-163
MPY.N	Wm*Wn,Acc,Wx,Wxd,Wy,Wyd	-(Multiply Wn by Wm) to Acc	1	1	5-165
MSC	Wm*Wn,Acc,Wx,Wxd,Wy, Wyd,AWB	Multiply and subtract from Acc	1	1	5-167
NEG	Acc	Negate Acc	1	1	5-183
SAC	Acc,#Slit4,Wd	Store Acc	1	1	5-216
SAC.R	Acc,#Slit4,Wd	Store rounded Acc	1	1	5-218
SFTAC	Acc,#Slit6	Arithmetic shift Acc by Slit6	1	1	5-223
SFTAC	Acc,Wn	Arithmetic shift Acc by (Wn)	1	1	5-224
SUB	Acc	Subtract accumulators	1	1	5-235

Section 4. Instruction Set Details

HIGHLIGHTS

This section of the manual contains the following major topics:

4.1 Data Addressing Modes

The dsPIC30F supports three native Addressing modes for accessing data memory, along with several forms of immediate addressing. Data accesses may be performed using file register, register direct or register indirect addressing, and immediate addressing allows a fixed value to be used by the instruction.

File register addressing provides the ability to operate on data stored in the lower 8K of data memory (Near RAM), and also move data between the working registers and the entire 64K data space. Register direct addressing is used to access the 16 memory mapped working registers, W0:W15. Register indirect addressing is used to efficiently operate on data stored in the entire 64K data space, using the contents of the working registers as an effective address. Immediate addressing does not access data memory, but provides the ability to use a constant value as an instruction operand. The address range of each mode is summarized in Table 4-1.

Table 4-1: dsPIC30F Addressing Modes

Addressing Mode	Address Range
File Register	`0x0000 - 0x1FFF` **(see Note)**
Register Direct	`0x0000 - 0x001F` (working register array W0:W15)
Register Indirect	`0x0000 - 0xFFFF`
Immediate	N/A (constant value)

Note: The address range for the File Register MOV is `0x0000 - 0xFFFE`.

4.1.1 File Register Addressing

File register addressing is used by instructions which use a predetermined data address as an operand for the instruction. The majority of instructions that support file register addressing provide access to the lower 8 Kbytes of data memory, which is called the Near RAM. However, the `MOV` instruction provides access to all 64 Kbytes of memory using file register addressing. This allows one to load data from any location in data memory to any working register, and store the contents of any working register to any location in data memory. It should be noted that file register addressing supports both byte and word accesses of data memory, with the exception of the `MOV` instruction, which accesses all 64K of memory as words. Examples of file register addressing are shown in Example 4-1.

Most instructions, which support file register addressing, perform an operation on the specified file register and the default working register WREG (see **Section 2.2.2 "Default Working Register (WREG)"**). If only one operand is supplied in the instruction, WREG is an implied operand and the operation results are stored back to the file register. In these cases, the instruction is effectively a read-modify-write instruction. However, when both the file register and WREG are specified in the instruction, the operation results are stored in WREG and the contents of the file register are unchanged. Sample instructions which show the interaction between the file register and WREG are shown in Example 4-2.

> **Note:** Instructions which support file register addressing use 'f' as an operand in the instruction summary tables of **Section 3. "Instruction Set Overview"**.

Example 4-1: File Register Addressing

```
DEC     0x1000         ; decrement data stored at 0x1000
```

Before Instruction:

```
  Data Memory 0x1000 = 0x5555
```

After Instruction:

```
  Data Memory 0x1000 = 0x5554
```

```
MOV     0x27FE, W0     ; move data stored at 0x27FE to W0
```

Before Instruction:

```
  W0 = 0x5555
  Data Memory 0x27FE = 0x1234
```

After Instruction:

```
  W0 = 0x1234
  Data Memory 0x27FE = 0x1234
```

Example 4-2: File Register Addressing and WREG

```
AND     0x1000         ; AND 0x1000 with WREG, store to 0x1000
```

Before Instruction:

```
  W0 (WREG) = 0x332C
  Data Memory 0x1000 = 0x5555
```

After Instruction:

```
  W0 (WREG) = 0x332C
  Data Memory 0x1000 = 0x1104
```

```
AND     0x1000, WREG   ; AND 0x1000 with WREG, store to WREG
```

Before Instruction:

```
  W0 (WREG) = 0x332C
  Data Memory 0x1000 = 0x5555
```

After Instruction:

```
  W0 (WREG) = 0x1104
  Data Memory 0x1000 = 0x5555
```

4.1.2 Register Direct Addressing

Register direct addressing is used to access the contents of the 16 working registers (W0:W15). The Register Direct Addressing mode is fully orthogonal, which allows any working register to be specified for any instruction which uses register direct addressing, and it supports both byte and word accesses. Instructions which employ register direct addressing use the contents of the specified working register as data to execute the instruction, so this Addressing mode is useful only when data already resides in the working register core. Sample instructions which utilize register direct addressing are shown in Example 4-3.

Another feature of register direct addressing is that it provides the ability for dynamic flow control. Since variants of the `DO` and `REPEAT` instruction support register direct addressing, one may generate flexible looping constructs using these instructions.

> **Note:** Instructions which must use register direct addressing, use the symbols Wb, Wn, Wns and Wnd in the summary tables of **Section 3. "Instruction Set Overview"**. Commonly, register direct addressing may also be used when register indirect addressing may be used. Instructions which use register indirect addressing, use the symbols Wd and Ws in the summary tables of **Section 3. "Instruction Set Overview"**.

Example 4-3: Register Direct Addressing

```
EXCH    W2, W3          ; Exchange W2 and W3
Before Instruction:
  W2 = 0x3499
  W3 = 0x003D
After Instruction:
  W2 = 0x003D
  W3 = 0x3499


IOR     #0x44, W0       ; Inclusive-OR 0x44 and W0
Before Instruction:
  W0 = 0x9C2E
After Instruction:
  W0 = 0x9C6E


SL      W6, W7, W8      ; Shift left W6 by W7, and store to W8
Before Instruction:
  W6 = 0x000C
  W7 = 0x0008
  W8 = 0x1234
After Instruction:
  W6 = 0x000C
  W7 = 0x0008
  W8 = 0x0C00
```

 Advance Information

4.1.3 Register Indirect Addressing

Register indirect addressing is used to access any location in data memory by treating the contents of a working register as an effective address (EA) to data memory. Essentially, the contents of the working register become a pointer to the location in data memory which is to be accessed by the instruction.

This Addressing mode is powerful, because it also allows one to modify the contents of the working register, either before or after the data access is made, by incrementing or decrementing the EA. By modifying the EA in the same cycle that an operation is being performed, register indirect addressing allows for the efficient processing of data that is stored sequentially in memory. The modes of indirect addressing supported by the dsPIC30F are shown in Table 4-2.

Table 4-2: Indirect Addressing Modes

Indirect Mode	Syntax	Function (Byte Instruction)	Function (Word Instruction)	Description
No Modification	[Wn]	EA = [Wn]	EA = [Wn]	The contents of Wn forms the EA.
Pre-Increment	[++Wn]	EA = [Wn+=1]	EA = [Wn+=2]	Wn is pre-incremented to form the EA.
Pre-Decrement	[--Wn]	EA = [Wn-=1]	EA = [Wn-=2]	Wn is pre-decremented to form the EA.
Post-Increment	[Wn++]	EA = [Wn]+= 1	EA = [Wn]+= 2	The contents of Wn forms he EA, then Wn is post-incremented.
Post-Decrement	[Wn--]	EA = [Wn]-= 1	EA = [Wn]-= 2	The contents of Wn forms the EA, then Wn is post-decremented.
Register Offset	[Wn+Wb]	EA = [Wn+Wb]	EA = [Wn+Wb]	The sum of Wn and Wb forms the EA. Wn and Wb are not modified.

Table 4-2 shows that four Addressing modes modify the EA used in the instruction, and this allows the following updates to be made to the working register: post-increment, post-decrement, pre-increment and pre-decrement. Since all EAs must be given as byte addresses, support is provided for Word mode instructions by scaling the EA update by 2. Namely, in Word mode, pre/post-decrements subtract 2 from the EA stored in the working register, and pre/post-increments add 2 to the EA. This feature ensures that after an EA modification is made, that the EA will point to the next adjacent word in memory. Example 4-4 shows how indirect addressing may be used to update the EA.

Table 4-2 also shows that the Register Offset mode addresses data which is offset from a base EA stored in a working register. This mode uses the contents of a second working register to form the EA by adding the two specified working registers. This mode does not scale for Word mode instructions, but offers the complete offset range of 64 Kbytes. Note that neither of the working registers used to form the EA are modified. Example 4-5 shows how register offset indirect addressing may be used to access data memory.

> **Note:** The `MOV` with offset instructions (pages page 151 and page 152) provides a literal addressing offset ability to be used with indirect addressing. In these instructions, the EA is formed by adding the contents of a working register to a signed 10-bit literal. Example 4-6 shows how these instructions may be used to move data to and from the working register array.

Example 4-4: Indirect Addressing with Effective Address Update

```
MOV.B   [W0++], [W13--]        ; byte move [W0] to [W13]
                               ; post-inc W0, post-dec W13
```

Before Instruction:

```
    W0 = 0x2300
    W13 = 0x2708
    Data Memory 0x2300 = 0x7783
    Data Memory 0x2708 = 0x904E
```

After Instruction:

```
    W0 = 0x2301
    W13 = 0x2707
    Data Memory 0x2300 = 0x7783
    Data Memory 0x2708 = 0x9083
```

```
ADD     W1, [--W5], [++W8]     ; pre-dec W5, pre-inc W8
                               ; add W1 to [W5], store in [W8]
```

Before Instruction:

```
    W1 = 0x0800
    W5 = 0x2200
    W8 = 0x2400
    Data Memory 0x21FE = 0x7783
    Data Memory 0x2402 = 0xAACC
```

After Instruction:

```
    W1 = 0x0800
    W5 = 0x21FE
    W8 = 0x2402
    Data Memory 0x21FE = 0x7783
    Data Memory 0x2402 = 0x7F83
```


Example 4-5: Indirect Addressing with Register Offset

```
MOV.B   [W0+W1], [W7++]        ; byte move [W0+W1] to W7, post-inc W7
Before Instruction:
    W0 = 0x2300
    W1 = 0x01FE
    W7 = 0x1000
    Data Memory 0x24FE = 0x7783
    Data Memory 0x1000 = 0x11DC
After Instruction:
    W0 = 0x2300
    W1 = 0x01FE
    W7 = 0x1001
    Data Memory 0x24FE = 0x7783
    Data Memory 0x1000 = 0x1183


LAC     [W0+W8], A             ; load ACCA with [W0+W8]
                               ; (sign-extend and zero-backfill)
Before Instruction:
    W0 = 0x2344
    W8 = 0x0008
    ACCA = 0x00 7877 9321
    Data Memory 0x234C = 0xE290
After Instruction:
    W0 = 0x2344
    W8 = 0x0008
    ACCA = 0xFF E290 0000
    Data Memory 0x234C = 0xE290
```

Example 4-6: Move with Literal Offset Instructions

```
MOV     [W0+0x20], W1          ; move [W0+0x20] to W1
Before Instruction:
    W0 = 0x1200
    W1 = 0x01FE
    Data Memory 0x1220 = 0xFD27
After Instruction:
    W0 = 0x1200
    W1 = 0xFD27
    Data Memory 0x1220 = 0xFD27


MOV     W4, [W8-0x300]         ; move W4 to [W8-0x300]
Before Instruction:
    W4 = 0x3411
    W8 = 0x2944
    Data Memory 0x2644 = 0xCB98
After Instruction:
    W4 = 0x3411
    W8 = 0x2944
    Data Memory 0x2644 = 0x3411
```

4.1.3.1 Register Indirect Addressing and the Instruction Set

The Addressing modes presented in Table 4-2 demonstrate the Indirect Addressing mode capability of the dsPIC30F. Due to operation encoding and functional considerations, not every instruction which supports indirect addressing supports all modes shown in Table 4-2. The majority of instructions which use indirect addressing support the No Modify, Pre-Increment, Pre-Decrement, Post-Increment and Post-Decrement Addressing modes. The `MOV` instructions, and several accumulator based DSP instructions, are also capable of using the Register Offset Addressing mode.

> **Note:** Instructions which use register indirect addressing use the operand symbols Wd and Ws in the summary tables of **Section 3. "Instruction Set Overview"**.

4.1.3.2 DSP `MAC` Indirect Addressing Modes

A special class of Indirect Addressing modes is utilized by the DSP `MAC` instructions. As is described later in **Section 4.14 "DSP MAC Instructions"**, the DSP `MAC` class of instructions are capable of performing two fetches from memory using effective addressing. Since DSP algorithms frequently demand a broader range of address updates, the Addressing modes offered by the DSP `MAC` instructions provide greater range in the size of the effective address update which may be made. Table 4-3 shows that both X and Y pre-fetches support Post-Increment and Post-Decrement Addressing modes, with updates of 2, 4 and 6 bytes. Since DSP instructions only execute in Word mode, no provisions are made for odd sized EA updates.

Table 4-3: DSP `MAC` Indirect Addressing Modes

Addressing Mode	X Memory	Y Memory
Indirect with no modification	EA = [Wx]	EA = [Wy]
Indirect with Post-Increment by 2	EA = [Wx]+= 2	EA = [Wy]+= 2
Indirect with Post-Increment by 4	EA = [Wx]+= 4	EA = [Wy]+= 4
Indirect with Post-Increment by 6	EA = [Wx]+= 6	EA = [Wy]+= 6
Indirect with Post-Decrement by 2	EA = [Wx]-= 2	EA = [Wy]-= 2
Indirect with Post-Decrement by 4	EA = [Wx]-= 4	EA = [Wy]-= 4
Indirect with Post-Decrement by 6	EA = [Wx]-= 6	EA = [Wy]-= 6
Indirect with Register Offset	EA = [W9 + W12]	EA = [W11 + W12]

Note: As described in **Section 4.14 "DSP MAC Instructions"**, only W8 and W9 may be used to access X Memory, and only W10 and W11 may be used to access Y Memory.

4.1.3.3 Modulo and Bit-Reversed Addressing Modes

The dsPIC30F provides support for two special Register Indirect Addressing modes, which are commonly used to implement DSP algorithms. Modulo (or circular) addressing provides an automated means to support circular data buffers in X and/or Y memory. Modulo buffers remove the need for software to perform address boundary checks, which can improve the performance of certain algorithms. Similarly, Bit-Reversed addressing allows one to access the elements of a buffer in a non-linear fashion. This Addressing mode simplifies data re-ordering for radix-2 FFT algorithms and provides a significant reduction in FFT processing time.

Both of these Addressing modes are powerful features of the dsPIC30F architecture, which can be exploited by any instruction that uses indirect addressing. Refer to the dsPIC30F MCU Family Reference Manual for details on using Modulo and Bit-Reversed addressing.

4.1.4 Immediate Addressing

In immediate addressing, the instruction encoding contains a predefined constant operand, which is used by the instruction. This Addressing mode may be used independently, but it is more frequently combined with the File Register, Direct and Indirect Addressing modes. The size of the immediate operand which may be used varies with the instruction type. Constants of size 1-bit (#lit1), 4-bit (#bit4, #lit4 and #Slit4), 5-bit (#lit5), 6-bit (#Slit6), 8-bit (#lit8), 10-bit (#lit10 and #Slit10), 14-bit (#lit14) and 16-bit (#lit16) may be used. Constants may be signed or unsigned and the symbols #Slit4, #Slit6 and #Slit10 designate a signed constant. All other immediate constants are unsigned. Table 4-4 shows the usage of each immediate operand in the instruction set.

Table 4-4: Immediate Operands in the Instruction Set

Operand	Instruction Usage
#lit1	PWRSAV
#bit4	BCLR, BSET, BTG, BTSC, BTSS, BTST, BTST.C, BTST.Z, BTSTS, BTSTS.C, BTSTS.Z
#lit4	ASR, LSR, SL
#Slit4	ADD, LAC, SAC, SAC.R
#lit5	ADD, ADDC, AND, CP, CPB, IOR, MUL.SU, MUL.UU, SUB, SUBB, SUBBR, SUBR, XOR
#Slit6	SFTAC
#lit8	MOV.B
#lit10	ADD, ADDC, AND, CP, CPB, IOR, RETLW, SUB, SUBB, XOR
#Slit10	MOV
#lit14	DISI, DO, LNK, REPEAT
#lit16	MOV

The syntax for immediate addressing requires that the number sign (#) must immediately precede the constant operand value. The "#" symbol indicates to the assembler that the quantity is a constant. If an out-of-range constant is used with an instruction, the assembler will generate an error. Several examples of immediate addressing are shown in Example 4-7.

Example 4-7: Immediate Addressing

```
PWRSAV  #1                 ; Enter IDLE mode


ADD.B   #0x10, W0          ; Add 0x10 to W0 (byte mode)

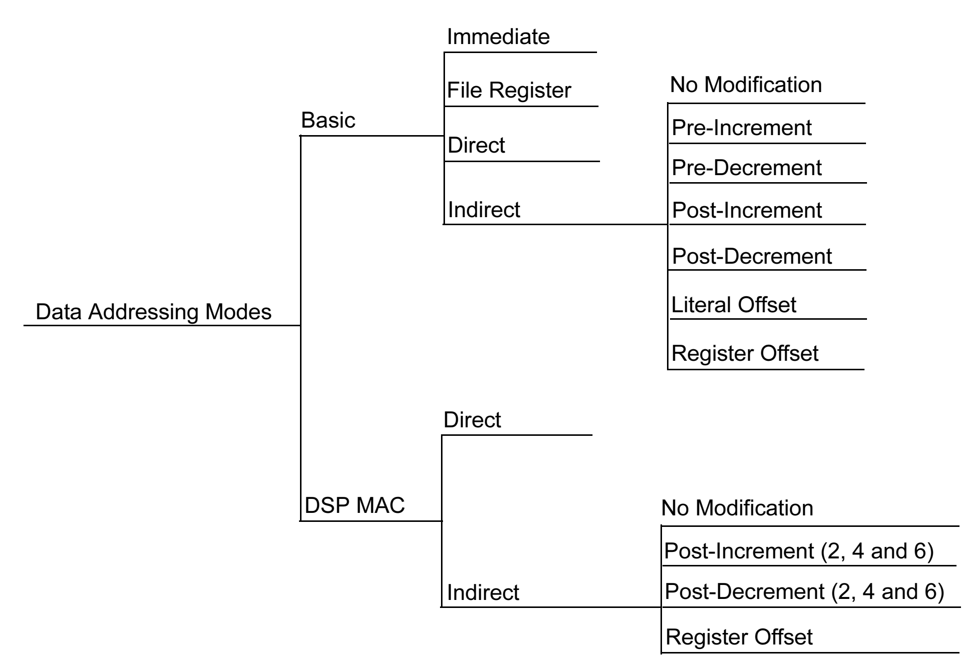
Before Instruction:
    W0 = 0x12A9
After Instruction:
    W0 = 0x12B9


XOR     W0, #1, [W1++]     ; Exclusive-OR W0 and 0x1
                           ; Store the result to [W1]
                           ; Post-increment W1
Before Instruction:
    W0 = 0xFFFF
    W1 = 0x0890
    Data Memory 0x0890 = 0x0032
After Instruction:
    W0 = 0xFFFF
    W1 = 0x0892
    Data Memory 0x0890 = 0xFFFE
```

4.1.5 Data Addressing Mode Tree

The Data Addressing modes of the dsPIC30F are summarized in Figure 4-1.

Figure 4-1: Data Addressing Mode Tree



4.2 Program Addressing Modes

The dsPIC30F has a 23-bit Program Counter (PC). The PC addresses the 24-bit wide program memory to fetch instructions for execution, and it may be loaded in several ways. For byte compatibility with the Table Read and Table Write instructions, each instruction word consumes two locations in program memory. This means that during serial execution, the PC is loaded with PC+2.

Several methods may be used to modify the PC in a non-sequential manner, and both absolute and relative changes may be made to the PC. The change to the PC may be from an immediate value encoded in the instruction, or a dynamic value contained in a working register. When DO looping is active, the PC is loaded with the address stored in the DOSTART register, after the instruction at the DOEND address is executed. For exception handling, the PC is loaded with the address of the exception handler, which is stored in the interrupt vector table. When required, the software stack is used to return scope to the foreground process from where the change in program flow occurred.

Table 4-5 summarizes the instructions which modify the PC of the dsPIC30F. When performing function calls, it is recommended that `RCALL` be used instead of `CALL`, since `RCALL` only consumes 1 word of program memory.

Table 4-5: Methods of Modifying Program Flow

Condition/Instruction	PC Modification	Software Stack Usage
Sequential Execution	PC = PC + 2	None
`BRA Expr`(1) (Branch Unconditionally)	PC = PC + 2*Slit16	None
`BRA Condition, Expr`(1) (Branch Conditionally)	PC = PC + 2 (condition false) PC = PC + 2*Slit16 (condition true)	None
`CALL Expr`(1) (Call Subroutine)	PC = lit23	PC+4 is pushed on the stack(2)
`CALL Wn` (Call Subroutine Indirect)	PC = Wn	PC+2 is pushed on the stack(2)
`GOTO Expr`(1) (Unconditional Jump)	PC = lit23	None
`GOTO Wn` (Unconditional Indirect Jump)	PC = Wn	None
`RCALL Expr`(1) (Relative Call)	PC = PC + 2*Slit16	PC+2 is pushed on the stack(2)
`RCALL Wn` (Computed Relative Call)	PC = PC + 2*Wn	PC+2 is pushed on the stack(2)
Exception Handling	PC = address of the exception handler (read from vector table)	PC+2 is pushed on the stack(3)
PC = Target `REPEAT` instruction (REPEAT Looping)	PC not modified (if `REPEAT` active)	None
PC = DOEND address (DO Looping)	PC = DOSTART (if `DO` active)	None

Note 1: For `BRA`, `CALL` and `GOTO`, the Expr may be a label, absolute address, or expression, which is resolved by the linker to a 16-bit or 23-bit value (Slit16 or lit23). See **Section 5. "Instruction Descriptions"** for details.

2: After `CALL` or `RCALL` is executed, `RETURN` or `RETLW` will pop the top-of-stack back into the PC.

3: After an exception is processed, `RETFIE` will pop the top-of-stack back into the PC.

4.3 Instruction Stalls

In order to maximize the data space EA calculation and operand fetch time, the X data space read and write accesses are partially pipelined. A consequence of this pipelining is that address register data dependencies may arise between successive read and write operations using common registers.

'Read After Write' (RAW) dependencies occur across instruction boundaries and are detected by the hardware. An example of a RAW dependency would be a write operation that modifies W5, followed by a read operation that uses W5 as an address pointer. The contents of W5 will not be valid for the read operation until the earlier write completes. This problem is resolved by stalling the instruction execution for one instruction cycle, which allows the write to complete before the next read is started.

4.3.1 RAW Dependency Detection

During the instruction pre-decode, the core determines if any address register dependency is imminent across an instruction boundary. The stall detection logic compares the W register (if any) used for the destination EA of the instruction currently being executed with the W register to be used by the source EA (if any) of the pre-fetched instruction. When a match between the destination and source registers is identified, a set of rules are applied to decide whether or not to stall the instruction by one cycle. Table 4-6 lists various RAW conditions which cause an instruction execution stall.

Table 4-6: Raw Dependency Rules (Detection By Hardware)

Destination Address Mode Using Wn	Source Address Mode Using Wn	Stall Required ?	Examples (Wn = W2)
Direct	Direct	No Stall	`ADD.W W0, W1, W2` `MOV.W W2, W3`
Indirect	Direct	No Stall	`ADD.W W0, W1, [W2]` `MOV.W W2, W3`
Indirect	Indirect	No Stall	`ADD.W W0, W1, [W2]` `MOV.W [W2], W3`
Indirect	Indirect with pre/post-modification	No Stall	`ADD.W W0, W1, [W2]` `MOV.W [W2++], W3`
Indirect with pre/post-modification	Direct	No Stall	`ADD.W W0, W1, [W2++]` `MOV.W W2, W3`
Direct	Indirect	Stall(1)	`ADD.W W0, W1, W2` `MOV.W [W2], W3`
Direct	Indirect with pre/post-modification	Stall(1)	`ADD.W W0, W1, W2` `MOV.W [W2++], W3`
Indirect	Indirect	Stall(1)	`ADD.W W0, W1, [W2]`(2) `MOV.W [W2], W3`(2)
Indirect	Indirect with pre/post-modification	Stall(1)	`ADD.W W0, W1, [W2]`(2) `MOV.W [W2++], W3`(2)
Indirect with pre/post-modification	Indirect	Stall(1)	`ADD.W W0, W1, [W2++]` `MOV.W [W2], W3`
Indirect with pre/post-modification	Indirect with pre/post-modification	Stall(1)	`ADD.W W0, W1, [W2++]` `MOV.W [W2++], W3`

Note 1: When stalls are detected, one cycle is added to the instruction execution time.
2: For these examples, the contents of W2 = the mapped address of W2 (`0x0004`).

4.3.2 Instruction Stalls and Exceptions

In order to maintain deterministic operation, instruction stalls are allowed to happen, even if they occur immediately prior to exception processing.

4.3.3 Instruction Stalls and Instructions that Change Program Flow

`CALL` and `RCALL` write to the stack using W15 and may, therefore, be subject to an instruction stall if the source read of the subsequent instruction uses W15.

`GOTO`, `RETFIE` and `RETURN` instructions are never subject to an instruction stall because they do not perform write operations to the working registers.

4.3.4 Instruction Stalls and DO/REPEAT Loops

Instructions operating in a DO or REPEAT loop are subject to instruction stalls, just like any other instruction. Stalls may occur on loop entry, loop exit and also during loop processing.

4.3.5 Instruction Stalls and PSV

Instructions operating in PSV address space are subject to instruction stalls, just like any other instruction. Should a data dependency be detected in the instruction immediately following the PSV data access, the second cycle of the instruction will initiate a stall. Should a data dependency be detected in the instruction immediately before the PSV data access, the last cycle of the previous instruction will initiate a stall.

> **Note:** Refer to the dsPIC30F Family Reference Manual for more detailed information about RAW instruction stalls.

4.4 Byte Operations

Since the dsPIC30F data memory is byte addressable, most of the base instructions may operate in either Byte mode or Word mode. When these instructions operate in Byte mode, the following rules apply:

- all direct working register references use the Least Significant Byte of the 16-bit working register and leave the Most Significant Byte unchanged
- all indirect working register references use the data byte specified by the 16-bit address stored in the working register
- all file register references use the data byte specified by the byte address
- the Status Register is updated to reflect the result of the byte operation

It should be noted that data addresses are always represented as **byte** addresses. Additionally, the native data format is little-endian, which means that words are stored with the Least Significant Byte at the lower address, and the Most Significant Byte at the adjacent, higher address (as shown in Figure 4-2). Example 4-8 shows sample byte move operations and Example 4-9 shows sample byte math operations.

> **Note:** Instructions which operate in Byte mode must use the "`.b`" or "`.B`" instruction extension to specify a byte instruction. For example, the following two instructions are valid forms of a byte clear operation:
>
> ```
> CLR.b  W0
> CLR.B  W0
> ```

Example 4-8: Sample Byte Move Operations

```
MOV.B   #0x30, W0      ; move the literal byte 0x30 to W0
```
Before Instruction:
```
W0 = 0x5555
```
After Instruction:
```
W0 = 0x5530
```

```
MOV.B   0x1000, W0     ; move the byte at 0x1000 to W0
```
Before Instruction:
```
W0 = 0x5555
Data Memory 0x1000 = 0x1234
```
After Instruction:
```
W0 = 0x5534
Data Memory 0x1000 = 0x1234
```

```
MOV.B   W0, 0x1001     ; byte move W0 to address 0x1001
```
Before Instruction:
```
W0 = 0x1234
Data Memory 0x1000 = 0x5555
```
After Instruction:
```
W0 = 0x1234
Data Memory 0x1000 = 0x3455
```

```
MOV.B   W0, [W1++]     ; byte move W0 to [W1], then post-inc W1
```
Before Instruction:
```
W0 = 0x1234
W1 = 0x1001
Data Memory 0x1000 = 0x5555
```
After Instruction:
```
W0 = 0x1234
W1 = 0x1002
Data Memory 0x1000 = 0x3455
```


Example 4-9: Sample Byte Math Operations

```
CLR.B   [W6--]             ; byte clear [W6], then post-dec W6
```

Before Instruction:

```
   W6 = 0x1001
   Data Memory 0x1000 = 0x5555
```

After Instruction:

```
   W6 = 0x1000
   Data Memory 0x1000 = 0x0055
```

```
SUB.B   W0, #0x10, W1      ; byte subtract literal 0x10 from W0
                           ; and store to W1
```

Before Instruction:

```
   W0 = 0x1234
   W1 = 0xFFFF
```

After Instruction:

```
   W0 = 0x1234
   W1 = 0xFF24
```

```
ADD.B   W0, W1, [W2++]     ; byte add W0 and W1, store to [W2]
                           ; and post-inc W2
```

Before Instruction:

```
   W0 = 0x1234
   W1 = 0x5678
   W2 = 0x1000
   Data Memory 0x1000 = 0x5555
```

After Instruction:

```
   W0 = 0x1234
   W1 = 0x5678
   W2 = 0x1001
   Data Memory 0x1000 = 0x55AC
```

4.5 Word Move Operations

Even though the dsPIC30F data space is byte addressable, all move operations made in Word mode must be word aligned. This means that for all source and destination operands, the Least Significant address bit must be '0'. If a word move is made to or from an odd address, an address error exception is generated. Likewise, all double-words must be word aligned. Figure 4-2 shows how bytes and words may be aligned in data memory. Example 4-10 contains several legal word move operations.

When an exception is generated due to a misaligned access, the exception is taken after the instruction executes. If the illegal access occurs from a data read, the operation will be allowed to complete, but the Least Significant bit of the source address will be cleared to force word alignment. If the illegal access occurs during a data write, the write will be inhibited. Example 4-11 contains several *illegal* word move operations.

Figure 4-2: Data Alignment in Memory

0x1001		**b0**	0x1000
0x1003	**b1**		0x1002
0x1005	**b3**	**b2**	0x1004
0x1007	**b5**	**b4**	0x1006
0x1009	**b7**	**b6**	0x1008
0x100B		**b8**	0x100A

Legend:
- b0 - byte stored at `0x1000`
- b1 - byte stored at `0x1003`
- b3:b2 - word stored at `0x1005:1004` (b2 is LSB)
- b7:b4 - double-word stored at `0x1009:0x1006` (b4 is LSB)
- b8 - byte stored at `0x100A`

Note: Instructions which operate in Word mode are not required to use an instruction extension. However, they may be specified with an optional "`.w`" or "`.W`" extension, if desired. For example, the following instructions are valid forms of a word clear operation:

```
CLR    W0
CLR.w  W0
CLR.W  W0
```


Example 4-10: Legal Word Move Operations

```
MOV     #0x30, W0          ; move the literal word 0x30 to W0
```

Before Instruction:

```
W0 = 0x5555
```

After Instruction:

```
W0 = 0x0030
```

```
MOV     0x1000, W0         ; move the word at 0x1000 to W0
```

Before Instruction:

```
W0 = 0x5555
Data Memory 0x1000 = 0x1234
```

After Instruction:

```
W0 = 0x1234
Data Memory 0x1000 = 0x1234
```

```
MOV     [W0], [W1++]       ; word move [W0] to [W1],
                           ; then post-inc W1
```

Before Instruction:

```
W0 = 0x1234
W1 = 0x1000
Data Memory 0x1000 = 0x5555
Data Memory 0x1234 = 0xAAAA
```

After Instruction:

```
W0 = 0x1234
W1 = 0x1002
Data Memory 0x1000 = 0xAAAA
Data Memory 0x1234 = 0xAAAA
```

Example 4-11: Illegal Word Move Operations

```
MOV     0x1001, W0          ; move the word at 0x1001 to W0
```

Before Instruction:

```
W0 = 0x5555
Data Memory 0x1000 = 0x1234
Data Memory 0x1002 = 0x5678
```

After Instruction:

```
W0 = 0x1234
Data Memory 0x1000 = 0x1234
Data Memory 0x1002 = 0x5678
```

ADDRESS ERROR TRAP GENERATED

(source address is misaligned, so `MOV` is performed)

```
MOV     W0, 0x1001          ; move W0 to the word at 0x1001
```

Before Instruction:

```
W0 = 0x1234
Data Memory 0x1000 = 0x5555
Data Memory 0x1002 = 0x6666
```

After Instruction:

```
W0 = 0x1234
Data Memory 0x1000 = 0x5555
Data Memory 0x1002 = 0x6666
```

ADDRESS ERROR TRAP GENERATED

(destination address is misaligned, so `MOV` is not performed)

```
MOV     [W0], [W1++]        ; word move [W0] to [W1],
                            ; then post-inc W1
```

Before Instruction:

```
W0 = 0x1235
W1 = 0x1000
Data Memory 0x1000 = 0x1234
Data Memory 0x1234 = 0xAAAA
Data Memory 0x1236 = 0xBBBB
```

After Instruction:

```
W0 = 0x1235
W1 = 0x1002
Data Memory 0x1000 = 0xAAAA
Data Memory 0x1234 = 0xAAAA
Data Memory 0x1236 = 0xBBBB
```

ADDRESS ERROR TRAP GENERATED

(source address is misaligned, so `MOV` is performed)

4.6 Using 10-bit Literal Operands

Several instructions which support Byte and Word mode have 10-bit operands. For byte instructions, a 10-bit literal is too large to use. So when 10-bit literals are used in Byte mode, the range of the operand must be reduced to 8-bits or the assembler will generate an error. Table 4-7 shows that the range of a 10-bit literal is 0:1023 in Word mode and 0:255 in Byte mode.

Instructions which employ 10-bit literals in Byte and Word mode are: `ADD`, `ADDC`, `AND`, `IOR`, `RETLW`, `SUB`, `SUBB` and `XOR`. Example 4-12 shows how positive and negative literals are used in Byte mode for the `ADD` instruction.

Table 4-7: 10-bit Literal Coding

Literal Value	Word Mode `kk kkkk kkkk`	Byte Mode `kkkk kkkk`
0	`00 0000 0000`	`0000 0000`
1	`00 0000 0001`	`0000 0001`
2	`00 0000 0010`	`0000 0010`
127	`00 0111 1111`	`0111 1111`
128	`00 1000 0000`	`1000 0000`
255	`00 1111 1111`	`1111 1111`
256	`01 0000 0000`	`N/A`
512	`10 0000 0000`	`N/A`
1023	`11 1111 1111`	`N/A`

Example 4-12: Using 10-bit Literals For Byte Operands

```
ADD.B   #0x80, W0       ; add 128 (or -128) to W0
ADD.B   #0x380, W0      ; ERROR... Illegal syntax for byte mode
ADD.B   #0xFF, W0       ; add 255 (or -1) to W0
ADD.B   #0x3FF, W0      ; ERROR... Illegal syntax for byte mode
ADD.B   #0xF, W0        ; add 15 to W0
ADD.B   #0x7F, W0       ; add 127 to W0
ADD.B   #0x100, W0      ; ERROR... Illegal syntax for byte mode
```

> **Note:** Using a literal value greater than 127 in Byte mode is functionally identical to using the equivalent negative two's complement value, since the Most Significant bit of the byte is set. When operating in Byte mode, the Assembler will accept either a positive or negative literal value (i.e., #-10).

4.7 Software Stack Pointer and Frame Pointer

4.7.1 Software Stack Pointer

The dsPIC30F features a software stack which facilitates function calls and exception handling. W15 is the default Stack Pointer (SP) and after any RESET, it is initialized to `0x0800`. This ensures that the SP will point to valid RAM in all dsPIC30F devices and permits stack availability for exceptions, which may occur before the SP is set by the user software. The user may reprogram the SP during initialization to any location within data space.

The SP always points to the first available free word (top-of-stack) and fills the software stack, working from lower addresses towards higher addresses. It pre-decrements for a stack pop (read) and post-increments for a stack push (write).

The software stack is manipulated using the `PUSH` and `POP` instructions. The `PUSH` and `POP` instructions are the equivalent of a `MOV` instruction, with W15 used as the destination pointer. For example, the contents of W0 can be pushed onto the top-of-stack (TOS) by

```
PUSH W0
```

This syntax is equivalent to

```
MOV W0,[W15++]
```

The contents of the TOS can be returned to W0 by

```
POP W0
```

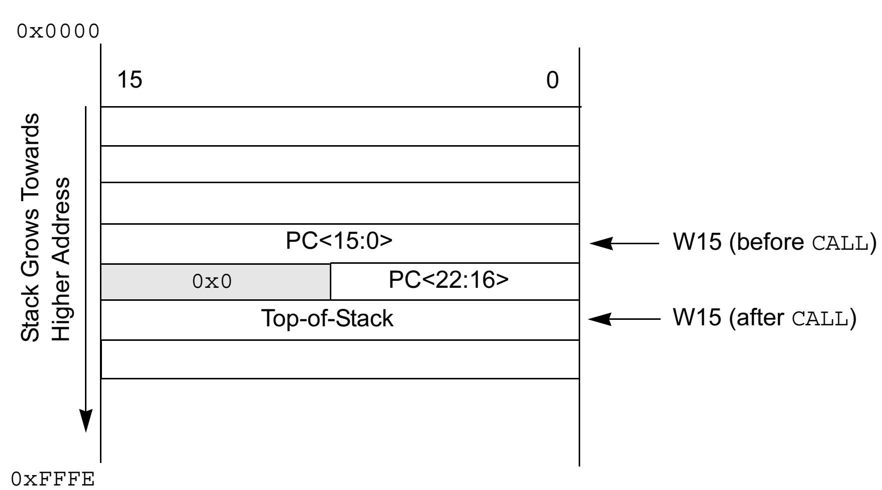
This syntax is equivalent to

```
MOV [--W15],W0
```

During any `CALL` instruction, the PC is pushed onto the stack, such that when the subroutine completes execution, program flow may resume from the correct location. When the PC is pushed onto the stack, PC<15:0> is pushed onto the first available stack word, then PC<22:16> is pushed. When PC<22:16> is pushed, the Most Significant 7 bits of the PC are zero-extended before the push is made, as shown in Figure 4-3. During exception processing, the Most Significant 7 bits of the PC are concatenated with the lower byte of the Status Register (SRL) and IPL<3>, CORCON<3>. This allows the primary Status Register contents and CPU Interrupt Priority Level to be automatically preserved during interrupts.

Note:	In order to protect against misaligned stack accesses, W15<0> is always clear.

Figure 4-3: Stack Operation for `CALL` Instruction



Note: For exceptions, the upper nine bits of the second pushed word contains the SRL and IPL<3>.

4.7.2 Stack Pointer Example

Figure 4-4 through Figure 4-7 show how the software stack is modified for the code snippet shown in Example 4-13. Figure 4-4 shows the software stack before the first `PUSH` has executed. Note that the SP has the initialized value of `0x0800`. Furthermore, the example loads `0x5A5A` and `0x3636` to W0 and W1, respectively. The stack is pushed for the first time in Figure 4-5 and the value contained in W0 is copied to TOS. W15 is automatically updated to point to the next available stack location, and the new TOS is `0x0802`. In Figure 4-6, the contents of W1 are pushed onto the stack, and the new TOS becomes `0x0804`. In Figure 4-7, the stack is popped, which copies the last pushed value (W1) to W3. The SP is decremented during the `POP` operation, and at the end of the example, the final TOS is `0x0802`.

Example 4-13: Stack Pointer Usage

```
MOV     #0x5A5A, W0     ; Load W0 with 0x5A5A
MOV     #0x3636, W1     ; Load W1 with 0x3636
PUSH    W0              ; Push W0 to TOS (see Figure 4-5)
PUSH    W1              ; Push W1 to TOS (see Figure 4-6)
POP     W3              ; Pop TOS to W3  (see Figure 4-7)
```

Figure 4-4: Stack Pointer Before The First `PUSH`

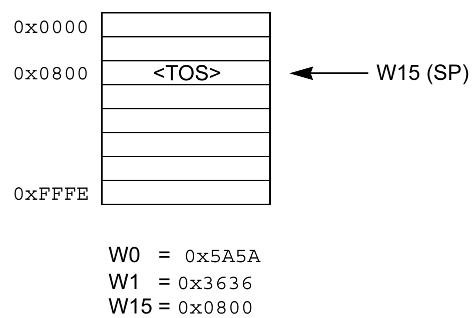


Figure 4-5: Stack Pointer After "`PUSH W0`" Instruction

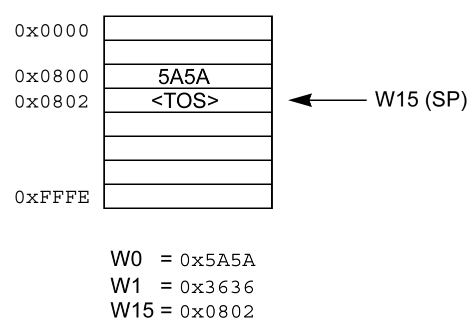


Figure 4-6: Stack Pointer After "`PUSH W1`" Instruction

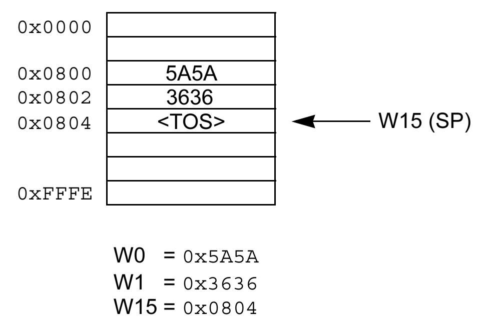
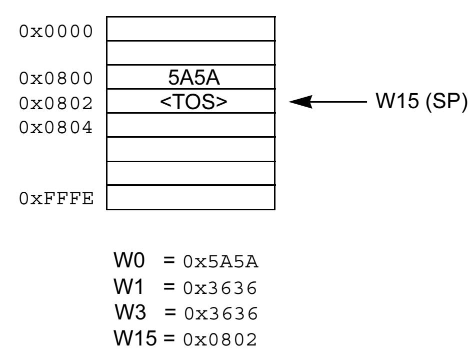


Figure 4-7: Stack Pointer After "`POP W3`" Instruction



Note: The contents of `0x802`, the new TOS, remain unchanged (`0x3636`).

4.7.3 Software Stack Frame Pointer

A stack frame is a user defined section of memory residing in the software stack. It is used to allocate memory for temporary variables which a function uses and one stack frame may be created for each function. W14 is the default Stack Frame Pointer (FP) and it is initialized to `0x0000` on any RESET. If the stack frame pointer is not used, W14 may be used like any other working register.

The link (`LNK`) and unlink (`ULNK`) instructions provide stack frame functionality. The `LNK` instruction is used to create a stack frame. It is used during a call sequence to adjust the SP, such that the stack may be used to store temporary variables utilized by the called function. After the function completes execution, the `ULNK` instruction is used to remove the stack frame created by the `LNK` instruction. The `LNK` and `ULNK` instructions must always be used together to avoid stack overflow.

4.7.4 Stack Frame Pointer Example

Figure 4-8 through Figure 4-10 show how a stack frame is created and removed for the code snippet shown in Example 4-14. Figure 4-8 shows the stack condition at the beginning of the example, before any registers are pushed to the stack. Here, W15 points to the first free stack location (TOS) and W14 points to a portion of stack memory allocated for the routine that is currently executing.

Before calling the function "COMPUTE", the parameters of the function (W0, W1 and W2) are pushed on the stack. After the "CALL COMPUTE" instruction is executed, the PC changes to the address of "COMPUTE" and the return address of the function "TASKA" is placed on the stack (Figure 4-9). Function "COMPUTE" then uses the "LNK #4" instruction to push the calling routine's frame pointer value onto the stack and the new frame pointer will be set to point to the current stack pointer. Then, the literal 4 is added to the stack pointer address in W15, which reserves memory for two words of temporary data (Figure 4-10).

Inside the function "COMPUTE", the FP is used to access the function parameters and temporary (local) variables. [W14+n] will access the temporary variables used by the routine and [W14-n] is used to access the parameters. At the end of the function, the ULNK instruction is used to copy the frame pointer address to the stack pointer and then pop the calling subroutine's frame pointer back to the W14 register. The ULNK instruction returns the stack back to the state shown in Figure 4-9.

A RETURN instruction will return to the code that called the subroutine. The calling code is responsible for removing the parameters from the stack. The RETURN and POP instructions restore the stack to the state shown in Figure 4-8.

Example 4-14: Frame Pointer Usage

```
TASKA:
    ...
    PUSH  W0       ; Push parameter 1
    PUSH  W1       ; Push parameter 2
    PUSH  W2       ; Push parameter 3
    CALL  COMPUTE  ; Call COMPUTE function
    POP   W2       ; Pop parameter 3
    POP   W1       ; Pop parameter 2
    POP   W0       ; Pop parameter 1
    ...

COMPUTE:
    LNK   #4       ; Stack FP, allocate 4 bytes for local variables
    ...
    ULNK           ; Free allocated memory, restore original FP
    RETURN         ; Return to TASKA
```

Figure 4-8: Stack at the Beginning of Example 4-14

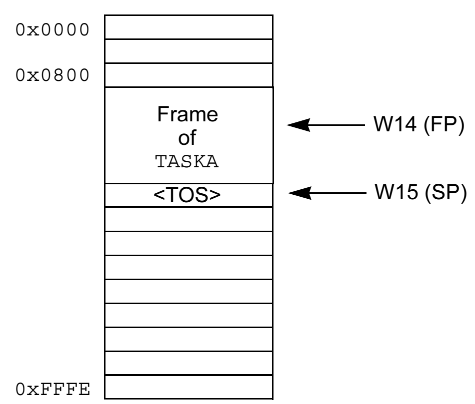


Figure 4-9: Stack After "CALL COMPUTE" Executes

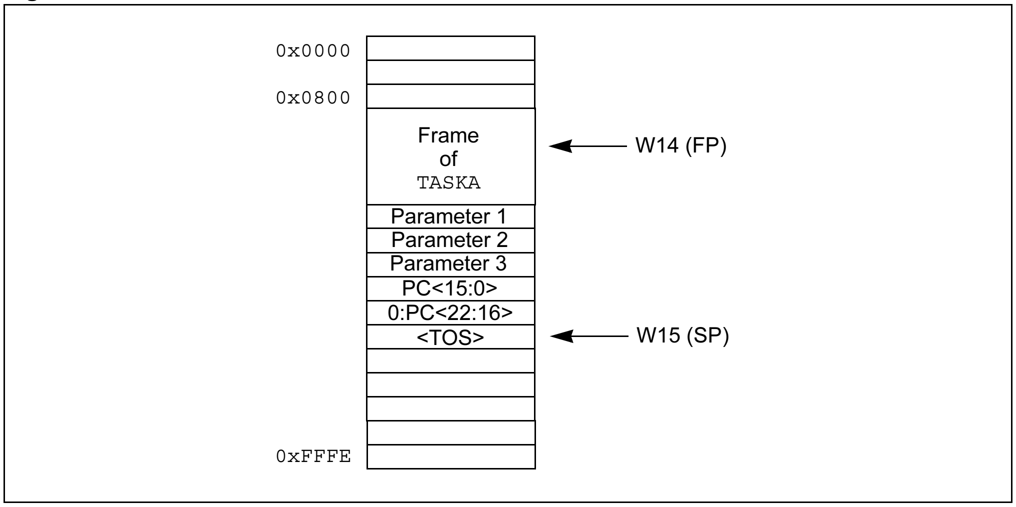
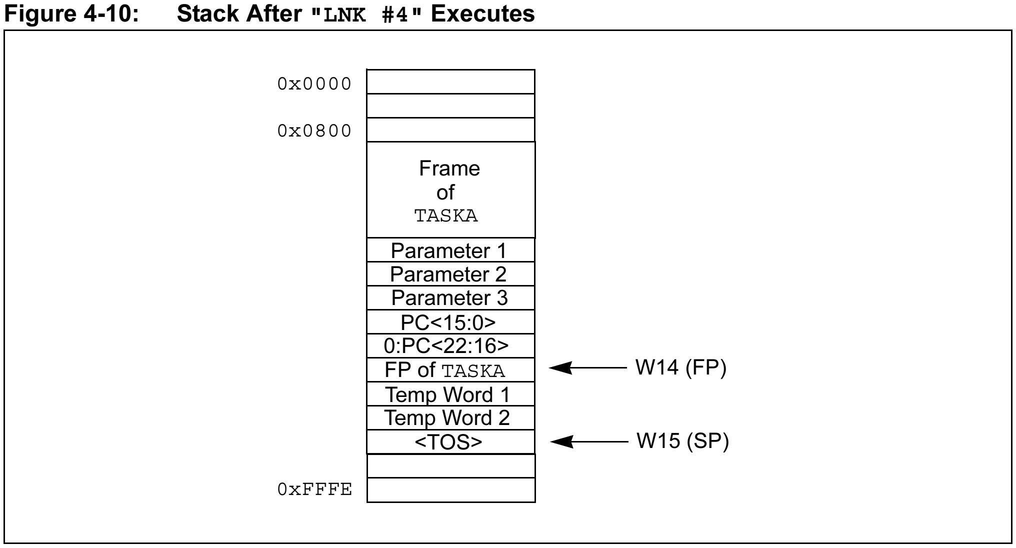


Figure 4-10: Stack After "LNK #4" Executes



4.7.5 Stack Pointer Overflow

There is a stack limit register (SPLIM) associated with the stack pointer that is reset to 0x0000. SPLIM is a 16-bit register, but SPLIM<0> is fixed to ‘0’, because all stack operations must be word aligned.

The stack overflow check will not be enabled until a word write to SPLIM occurs, after which time it can only be disabled by a device RESET. All effective addresses generated using W15 as a source or destination are compared against the value in SPLIM. Should the effective address be greater than the contents of SPLIM, then a stack error trap is generated.

If stack overflow checking has been enabled, a stack error trap will also occur if the W15 effective address calculation wraps over the end of data space (0xFFFF).

Refer to the dsPIC30F Family Reference Manual for more information on the stack error trap.

4.7.6 Stack Pointer Underflow

The stack is initialized to 0x0800 during RESET. A stack error trap will be initiated should the stack pointer address ever be less than 0x0800.

Note:	Locations in data space between 0x0000 and 0x07FF are, in general, reserved for core and peripheral special function registers.

4.8 Conditional Branch Instructions

Conditional branch instructions are used to direct program flow, based on the contents of the Status Register. These instructions are generally used in conjunction with a Compare class instruction, but they may be employed effectively after any operation that modifies the Status Register.

The compare instructions `CP`, `CP0` and `CPB`, perform a subtract operation (minuend - subtrahend), but do not actually store the result of the subtraction. Instead, compare instructions just update the flags in the Status Register, such that an ensuing conditional branch instruction may change program flow by testing the contents of the updated Status Register. If the result of the Status Register test is true, the branch is taken. If the result of the Status Register test is false, the branch is not taken.

The conditional branch instructions supported by the dsPIC30F devices are shown in Table 4-8. This table identifies the condition in the Status Register which must be true for the branch to be taken. In some cases, just a single bit is tested (as in `BRA C`), while in other cases, a complex logic operation is performed (as in `BRA GT`). It is worth noting that both signed and unsigned conditional tests are supported, and that support is provided for DSP algorithms with the OA, OB, SA and SB condition mnemonics.

Table 4-8: Conditional Branch Instructions

Condition Mnemonic(1)	Description	Status Test
C	Carry (not Borrow)	C
GE	Signed greater than or equal	$(\overline{N}\&\&\overline{OV}) \vert\vert (N\&\&OV)$
GEU(2)	Unsigned greater than or equal	C
GT	Signed greater than	$(\overline{Z}\&\&\overline{N}\&\&\overline{OV}) \vert\vert (\overline{Z}\&\&N\&\&OV)$
GTU	Unsigned greater than	$C\&\&\overline{Z}$
LE	Signed less than or equal	$Z \vert\vert (\overline{N}\&\&OV) \vert\vert (N\&\&\overline{OV})$
LEU	Unsigned less than or equal	$\overline{C} \vert\vert Z$
LT	Signed less than	$(\overline{N}\&\&OV) \vert\vert (N\&\&\overline{OV})$
LTU(3)	Unsigned less than	$\overline{C}$
N	Negative	N
NC	Not Carry (Borrow)	$\overline{C}$
NN	Not Negative	$\overline{N}$
NOV	Not Overflow	$\overline{OV}$
NZ	Not Zero	$\overline{Z}$
OA	Accumulator A overflow	OA
OB	Accumulator B overflow	OB
OV	Overflow	OV
SA	Accumulator A saturate	SA
SB	Accumulator B saturate	SB
Z	Zero	Z

Note 1: Instructions are of the form: `BRA` mnemonic, Expr.
2: GEU is identical to C and will reverse assemble to `BRA C`, Expr.
3: LTU is identical to NC and will reverse assemble to `BRA NC`, Expr.

> **Note:** The "Compare and Skip" instructions (`CPSEQ`, `CPSGT`, `CPSLT`, `CPSNE`) do not modify the Status Register.

4.9 Z Status Bit

The Z status bit is a special zero status bit that is useful for extended precision arithmetic. The Z bit functions like a normal Z flag for all instructions, except those that use the carry/borrow input (ADDC, CPB, SUBB and SUBBR). For the ADDC, CPB, SUBB and SUBBR instructions, the Z bit can only be cleared and never set. If the result of one of these instructions is non-zero, the Z bit will be cleared and will remain cleared, *regardless of the result of subsequent ADDC, CPB, SUBB or SUBBR operations*. This allows the Z bit to be used for performing a simple zero check on the result of a series of extended precision operations.

A sequence of instructions working on multi-precision data (starting with an instruction with no carry/borrow input) will automatically logically AND the successive results of the zero test. All results must be zero for the Z flag to remain set at the end of the sequence of operations. If the result of the ADDC, CPB, SUBB or SUBBR instruction is non-zero, the Z bit will be cleared and remain cleared for all subsequent ADDC, CPB, SUBB or SUBBR instructions. Example 4-15 shows how the Z bit operates for a 32-bit addition. It shows how the Z bit is affected for a 32-bit addition implemented with an ADD/ADDC instruction sequence. The first example generates a zero result for only the MSWord, and the second example generates a zero result for both the LSWord and MSWord.

Example 4-15: 'Z' Status bit Operation for 32-bit Addition

```
; Add two doubles (W0:W1 and W2:W3)
; Store the result in W5:W4
ADD     W0, W2, W4     ; Add LSWord and store to W4
ADDC    W1, W3, W5     ; Add MSWord and store to W5
```

Before 32-bit Addition (zero result for MSWord):

```
W0 = 0x2342
W1 = 0xFFF0
W2 = 0x39AA
W3 = 0x0010
W4 = 0x0000
W5 = 0x0000
SR = 0x0000
```

After 32-bit Addition:

```
W0 = 0x2342
W1 = 0xFFF0
W2 = 0x39AA
W3 = 0x0010
W4 = 0x5CEC
W5 = 0x0000
SR = 0x0201 (DC,C=1)
```

Before 32-bit Addition (zero result for LSWord and MSWord):

```
W0 = 0xB76E
W1 = 0xFB7B
W2 = 0x4892
W3 = 0x0484
W4 = 0x0000
W5 = 0x0000
SR = 0x0000
```

After 32-bit Addition:

```
W0 = 0xB76E
W1 = 0xFB7B
W2 = 0x4892
W3 = 0x0485
W4 = 0x0000
W5 = 0x0000
SR = 0x0103 (DC,Z,C=1)
```


4.10 Assigned Working Register Usage

The 16 working registers of the dsPIC30F provide a large register set for efficient code generation and algorithm implementation. In an effort to maintain an instruction set that provides advanced capability, a stable run-time environment and backwards compatibility with earlier Microchip processor cores, some working registers have a pre-assigned usage. Table 4-9 summarizes these working register assignments, with details provided in subsections **Section 4.10.1 "Implied DSP Operands"** through **Section 4.10.3 "PICmicro® Microcontroller Compatibility"**.

Table 4-9: Special Working Register Assignments

Register	Special Assignment
W0	Default WREG, Divide Quotient
W1	Divide Remainder
W2	"MUL f" Product Least Significant Word
W3	"MUL f" Product Most Significant Word
W4	MAC Operand
W5	MAC Operand
W6	MAC Operand
W7	MAC Operand
W8	MAC Pre-fetch Address (X Memory)
W9	MAC Pre-fetch Address (X Memory)
W10	MAC Pre-fetch Address (Y Memory)
W11	MAC Pre-fetch Address (Y Memory)
W12	MAC Pre-fetch Offset
W13	MAC Write Back Destination
W14	Frame Pointer
W15	Stack Pointer

4.10.1 Implied DSP Operands

To assist instruction encoding and maintain uniformity among the DSP class of instructions, some working registers have pre-assigned functionality. For all DSP instructions which have pre-fetch ability, the following 10 register assignments must be adhered to:

- W4-W7 are used for arithmetic operands
- W8-W11 are used for pre-fetch addresses (pointers)
- W12 is used for the pre-fetch register offset index
- W13 is used for the accumulator write back destination

These restrictions only apply to the DSP MAC class of instructions, which utilize working registers and have pre-fetch ability (described in **Section 4.15 "DSP Accumulator Instructions"**). The affected instructions are `CLR, ED, EDAC, MAC, MOVSAC, MPY, MPY.N` and `MSC`.

The DSP Accumulator class of instructions (described in **Section 4.15 "DSP Accumulator Instructions"**) are not required to follow the working register assignments in Table 4-9 and may freely use any working register when required.

4.10.2 Implied Frame and Stack Pointer

To accommodate software stack usage, W14 is the implied frame pointer (used by the `LNK` and `ULNK` instructions) and W15 is the implied stack pointer (used by the `CALL, LNK, POP, PUSH, RCALL, RETFIE, RETLW, RETURN, TRAP` and `ULNK` instructions). Even though W14 and W15 have this implied usage, they may still be used as generic operands in any instruction, with the exceptions outlined in **Section 4.10.1 "Implied DSP Operands"**. If W14 and W15 must be used for other purposes (it is strongly advised that they remain reserved for the Frame and Stack pointer), extreme care must be taken such that the run-time environment is not corrupted.

4.10.3 PICmicro® Microcontroller Compatibility

4.10.3.1 Default Working Register WREG

To ease the migration path for users of the Microchip PICmicro families, the dsPIC30F has matched the functionality of the PICmicro instruction sets as closely as possible. One major difference between the dsPIC30F and the PICmicro processors is the number of working registers provided. The PICmicro families only provide one 8-bit working register, while the dsPIC30F provides sixteen, 16-bit working registers. To accommodate for the one working register of the PICmicro MCU, the dsPIC30F instruction set has designated one working register to be the default working register for all legacy file register instructions. The default working register is set to W0, and it is used by all instructions which use file register addressing.

Additionally, the syntax used by the dsPIC30F assembler to specify the default working register is similar to that used by the PICmicro assembler. As shown in the detailed instruction descriptions in **Section 5. "Instruction Descriptions"**, "WREG" must be used to specify the default working register. Example 4-16 shows several instructions which use WREG.

Example 4-16: Using the Default Working Register WREG

```
ADD     RAM100          ; add RAM100 and WREG, store in RAM100
ASR     RAM100, WREG    ; shift RAM100 right, store in WREG
CLR.B   WREG            ; clear the WREG LS Byte
DEC     RAM100, WREG    ; decrement RAM100, store in WREG
MOV     WREG, RAM100    ; move WREG to RAM100
SETM    WREG            ; set all bits in the WREG
XOR     RAM100          ; XOR RAM100 and WREG, store in RAM100
```

4.10.3.2 PRODH:PRODL Register Pair

Another significant difference between the Microchip PICmicro and dsPIC30F architectures is the multiplier. Some PICmicro families support an 8-bit x 8-bit multiplier, which places the multiply product in the PRODH:PRODL register pair. The dsPIC30F has a 17-bit x 17-bit multiplier, which may place the result into any two successive working registers (starting with an even register), or an accumulator.

Despite this architectural difference, the dsPIC30F still supports the legacy file register multiply instruction (`MULWF`) with the "`MUL{.B} f`" instruction (described on page 5-169). Supporting the legacy `MULWF` instruction has been accomplished by mapping the PRODH:PRODL registers to the working register pair W3:W2. This means that when "`MUL{.B} f`" is executed in Word mode, the multiply generates a 32-bit product which is stored in W3:W2, where W3 has the Most Significant Word of the product and W2 has the Least Significant Word of the product. When "`MUL{.B} f`" is executed in Byte mode, the 16-bit product is stored in W2, and W3 is unaffected. Examples of this instruction are shown in Example 4-17.

Example 4-17: Unsigned f and WREG Multiply (Legacy `MULWF` Instruction)

```
MUL.B   0x100      ; (0x100)*WREG (byte mode), store to W2
```

Before Instruction:

```
W0 (WREG) = 0x7705
W2 = 0x1235
W3 = 0x1000
Data Memory 0x0100 = 0x1255
```

After Instruction:

```
W0 (WREG) = 0x7705
W2 = 0x01A9
W3 = 0x1000
Data Memory 0x0100 = 0x1255
```

```
MUL     0x100      ; (0x100)*WREG (word mode), store to W3:W2
```

Before Instruction:

```
W0 (WREG) = 0x7705
W2 = 0x1235
W3 = 0x1000
Data Memory 0x0100 = 0x1255
```

After Instruction:

```
W0 (WREG) = 0x7705
W2 = 0xDEA9
W3 = 0x0885
Data Memory 0x0100 = 0x1255
```

4.11 DSP Data Formats

4.11.1 Integer and Fractional Data

The dsPIC30F devices support both integer and fractional data types. Integer data is inherently represented as a signed two's complement value, where the Most Significant bit is defined as a sign bit. Generally speaking, the range of an N-bit two's complement integer is -2^{N-1} to $2^{N-1} - 1$. For a 16-bit integer, the data range is -32768 (`0x8000`) to 32767 (`0x7FFF`), including `0`. For a 32-bit integer, the data range is -2,147,483,648 (`0x8000 0000`) to 2,147,483,647 (`0x7FFF FFFF`).

Fractional data is represented as a two's complement number, where the Most Significant bit is defined as a sign bit, and the radix point is implied to lie just after the sign bit. This format is commonly referred to as 1.15 (or Q15) format, where 1 is the number of bits used to represent the integer portion of the number, and 15 is the number of bits used to represent the fractional portion. The range of an N-bit two's complement fraction with this implied radix point is -1.0 to $(1 - 2^{1-N})$. For a 16-bit fraction, the 1.15 data range is -1.0 (`0x8000`) to 0.999969482 (`0x7FFF`), including 0.0 and it has a precision of 3.01518×10^{-5}. In Normal Saturation mode, the 32-bit accumulators use a 1.31 format, which enhances the precision to 4.6566×10^{-10}.

Super Saturation mode expands the dynamic range of the accumulators by using the 8 bits of the Upper Accumulator register (ACCxU) as guard bits. Guard bits are used if the value stored in the accumulator overflows beyond the 32nd bit, and they are useful for implementing DSP algorithms. This mode is enabled when the **ACCSAT** bit (CORCON<4>), is set to '1' and it expands the accumulators to 40-bits. The accumulators then support an integer range of -5.498×10^{11} (`0x80 0000 0000`) to 5.498×10^{11} (`0x7F FFFF FFFF`). In Fractional mode, the guard bits of the accumulator do not modify the location of the radix point and the 40-bit accumulators use a 9.31 fractional format. Note that all fractional operation results are stored in the 40-bit accumulator, justified with a 1.31 radix point. As in Integer mode, the guard bits merely increase the dynamic range of the accumulator. 9.31 fractions have a range of -256.0 (`0x80 0000 0000`) to $(256.0 - 4.65661 \times 10^{-10})$ (`0x7F FFFF FFFF`). Table 4-10 identifies the range and precision of integers and fractions on the dsPIC30F devices for 16-bit, 32-bit and 40-bit registers.

It should be noted that, with the exception of DSP multiplies, the dsPIC30F ALU operates identically on integer and fractional data. Namely, an addition of two integers will yield the same result (binary number) as the addition of two fractional numbers. The only difference is how the result is interpreted by the user. However, multiplies performed by DSP operations are different. In these instructions, data format selection is made by the **IF** bit, CORCON<0>, and it must be set accordingly ('`0`' for Fractional mode, '`1`' for Integer mode). This is required because of the implied radix point used by dsPIC30F fractions. In Integer mode, multiplying two 16-bit integers produces a 32-bit integer result. However, multiplying two 1.15 values generates a 2.30 result. Since the dsPIC30F devices use 1.31 format for the accumulators, a DSP multiply in Fractional mode also includes a left shift of one bit to keep the radix point properly aligned. This feature reduces the resolution of the DSP multiplier to 2^{-30}, but has no other effect on the computation (e.g., 0.5 x 0.5 = 0.25).

Table 4-10: dsPIC30F Data Ranges

Register Size	Integer Range	Fraction Range	Fraction Resolution
16-bit	-32768 to 32767	-1.0 to $(1.0 - 2^{-15})$	3.052×10^{-5}
32-bit	-2,147,483,648 to 2,147,483,647	-1.0 to $(1.0 - 2^{-31})$	4.657×10^{-10}
40-bit	-549,755,813,888 to 549,755,813,887	-256.0 to $(256.0 - 2^{-31})$	4.657×10^{-10}

4.11.2 Integer and Fractional Data Representation

Having a working knowledge of how integer and fractional data are represented on the dsPIC30F is fundamental to working with the device. Both integer and fractional data treat the Most Significant bit as a sign bit, and the binary exponent decreases by one as the bit position advances towards the Least Significant bit. The binary exponent for an N-bit integer starts at (N-1) for the Most Significant bit, and ends at `0` for the Least Significant bit. For an N-bit fraction, the binary exponent starts at `0` for the Most Significant bit, and ends at (`1-N`) for the Least Significant bit. This is shown in Figure 4-11 for a positive value and in Figure 4-12 for a negative value.

Converting between integer and fractional representations can be performed using simple division and multiplication. To go from an N-bit integer to a fraction, divide the integer value by 2^{N-1}. Likewise, to convert an N-bit fraction to an integer, multiply the fractional value by 2^{N-1}.

Figure 4-11: Different Representations of `0x4001`

Integer:

0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

-2^{15} 2^{14} 2^{13} 2^{12} . . . . . . 2^{0}

`0x4001` $= 2^{14} + 2^{0} = 16384 + 1 =$ **16385**

1.15 Fractional:

0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

-2^{0} . 2^{-1} 2^{-2} 2^{-3} . . . . . . 2^{-15}

Implied Radix Point

`0x4001` $= 2^{-1} + 2^{-15} = 0.5 + .000030518 =$ **0.500030518**

Figure 4-12: Different Representations of `0xC002`

Integer:

1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

-2^{15} 2^{14} 2^{13} 2^{12} . . . . . . 2^{0}

`0xC002` $= -2^{15} + 2^{14} + 2^{1} = -32768 + 16384 + 2 =$ **-16382**

1.15 Fractional:

1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

-2^{0} . 2^{-1} 2^{-2} 2^{-3} . . . . . . 2^{-15}

Implied Radix Point

`0xC002` $= -2^{0} + 2^{-1} + 2^{-14} = -1.0 + 0.5 + 0.000061035 =$ **-0.499938965**

4.12 Accumulator Usage

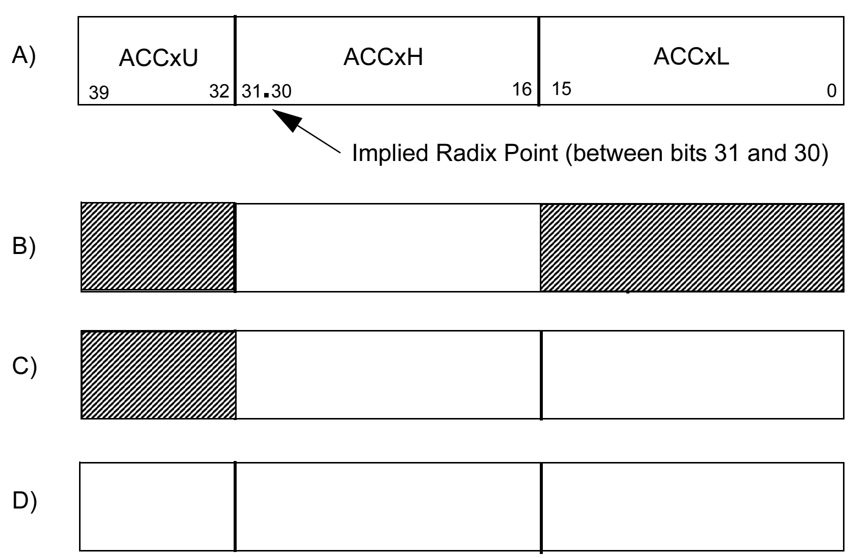
Accumulators A and B are utilized by DSP instructions to perform mathematical and shifting operations. Since the accumulators are 40-bits wide and the X and Y data paths are only 16-bits, the method to load and store the accumulators must be understood.

Item A in Figure 4-13 shows that each 40-bit accumulator (ACCA and ACCB) consists of an 8-bit Upper register (ACCxU), a 16-bit High register (ACCxH) and a 16-bit Low register (ACCxL). To address the bus alignment requirement and provide the ability for 1.31 math, ACCxH is used as a destination register for loading the accumulator (with the `LAC` instruction), and also as a source register for storing the accumulator (with the `SAC.R` instruction). This is represented by Item B, Figure 4-13, where the upper and lower portions of the accumulator are shaded. In reality, during accumulator loads, ACCxL is zero backfilled and ACCxU is sign-extended to represent the sign of the value loaded in ACCxH.

When Normal (31-bit) Saturation is enabled, DSP operations (such as `ADD`, `MAC`, `MSC`, etc.) utilize solely ACCxH:ACCxL (Item C in Figure 4-13) and ACCxU is only used to maintain the sign of the value stored in ACCxH:ACCxL. For instance, when a `MPY` instruction is executed, the result is stored in ACCxH:ACCxL, and the sign of the result is extended through ACCxU.

When Super Saturation is enabled, all registers of the accumulator may be used (Item D in Figure 4-13) and the results of DSP operations are stored in ACCxU:ACCxH:ACCxL. The benefit of ACCxU is that it increases the dynamic range of the accumulator, as described in **Section 4.11.1 "Integer and Fractional Data"**. Refer to Table 4-10 to see the range of values which may be stored in the accumulator when in Normal and Super Saturation modes.

Figure 4-13: Accumulator Alignment and Usage



A) 40-bit Accumulator consists of ACCxU:ACCxH:ACCxL
B) Load and Store operations
C) Operations in Normal Saturation mode
D) Operations in Super Saturation mode

4.13 Accumulator Access

The six registers of Accumulator A and Accumulator B are memory mapped like any other special function register. This feature allows them to be accessed with file register or indirect addressing, using any instruction which supports such addressing. However, it is recommended that the DSP instructions `LAC`, `SAC` and `SAC.R` be used to load and store the accumulators, since they provide sign-extension, shifting and rounding capabilities. `LAC`, `SAC` and `SAC.R` instruction details are provided in **Section 5. “Instruction Descriptions”**.

Note:	For convenience, ACCAU and ACCBU are sign-extended to 16-bits. This provides the flexibility to access these registers using either Byte or Word mode (when file register or indirect addressing is used).

4.14 DSP `MAC` Instructions

The DSP Multiply and Accumulate (`MAC`) operations are a special suite of instructions which provide the most efficient use of the dsPIC30F architecture. The DSP `MAC` instructions, shown in Table 4.14, utilize both the X and Y data paths of the CPU core, which enables these instructions to perform the following operations all in one cycle:

- two reads from data memory using pre-fetch working registers (`MAC` Pre-fetches)
- two updates to pre-fetch working registers (`MAC` Pre-fetch Register Updates)
- one mathematical operation with an accumulator (`MAC` Operations)

In addition, four of the ten DSP `MAC` instructions are also capable of performing an operation with one accumulator, while storing out the rounded contents of the alternate accumulator. This feature is called Accumulator Write Back (WB) and it provides flexibility for the software developer. For instance, the Accumulator WB may be used to run two algorithms concurrently, or efficiently process complex numbers, among other things.

Table 4-11: DSP `MAC` Instructions

Instruction	Description	Accumulator WB?
`CLR`	Clear accumulator	Yes
`ED`	Euclidean distance (no accumulate)	No
`EDAC`	Euclidean distance	No
`MAC`	Multiply and accumulate	Yes
`MAC`	Square and accumulate	No
`MOVSAC`	Move from X and Y bus	Yes
`MPY`	Multiply to accumulator	No
`MPY`	Square to accumulator	No
`MPY.N`	Negative multiply to accumulator	No
`MSC`	Multiply and subtract	Yes

4.14.1 `MAC` Pre-Fetches

Pre-Fetches (or data reads) are made using the effective address stored in the working register. The two pre-fetches from data memory must be specified using the working registers assignments shown in Table 4-9. One read must occur from the X data bus using W8 or W9, and one read must occur from the Y data bus using W10 or W11. Allowable destination registers for both pre-fetches are W4-W7.

As shown in Table 4-3, one special Addressing mode exists for the `MAC` class of instructions. This mode is the Register Offset Addressing mode and utilizes W12. In this mode, the pre-fetch is made using the effective address of the specified working register, plus the 16-bit signed value stored in W12. Register Offset Addressing may only be used in the X space with W9, and in the Y-space with W11.

4
Instruction Set Details

4.14.2 MAC Pre-Fetch Register Updates

After the MAC pre-fetches are made, the effective address stored in each pre-fetch working register may be modified. This feature enables efficient single cycle processing for data stored sequentially in X and Y memory. Since all DSP instructions execute in Word mode, only even numbered updates may be made to the effective address stored in the working register. Allowable address modifications to each pre-fetch register are -6, -4, -2, 0 (no update), +2, +4 and +6. This means that effective address updates may be made up to 3 words in either direction.

When the Register Offset Addressing mode is used, no update is made to the base pre-fetch register (W9 or W11), or the offset register (W12).

4.14.3 MAC Operations

The mathematical operations performed by the MAC class of DSP instructions center around multiplying the contents of two working registers and either adding or storing the result to either Accumulator A or Accumulator B. This is the operation of the MAC, MPY, MPY.N and MSC instructions. Table 4-9 shows that W4-W7 must be used for data source operands in the MAC class of instructions. W4-W7 may be combined in any fashion, and when the same working register is specified for both operands, a square or square and accumulate operation is performed.

For the ED and EDAC instructions, the same multiplicand operand *must* be specified by the instruction, because this is the definition of the Euclidean Distance operation. Another unique feature about this instruction is that the values pre-fetched from X and Y memory are not actually stored in W4-W7. Instead, only the *difference* of the pre-fetched data words is stored in W4-W7.

The two remaining MAC class instructions, CLR and MOVSAC, are useful for initiating or completing a series of MAC or EDAC instructions and do not use the multiplier. CLR has the ability to clear Accumulator A or B, pre-fetch two values from data memory and store the contents of the other accumulator. Similarly, MOVSAC has the ability to pre-fetch two values from data memory and store the contents of either accumulator.

4.14.4 MAC Write Back

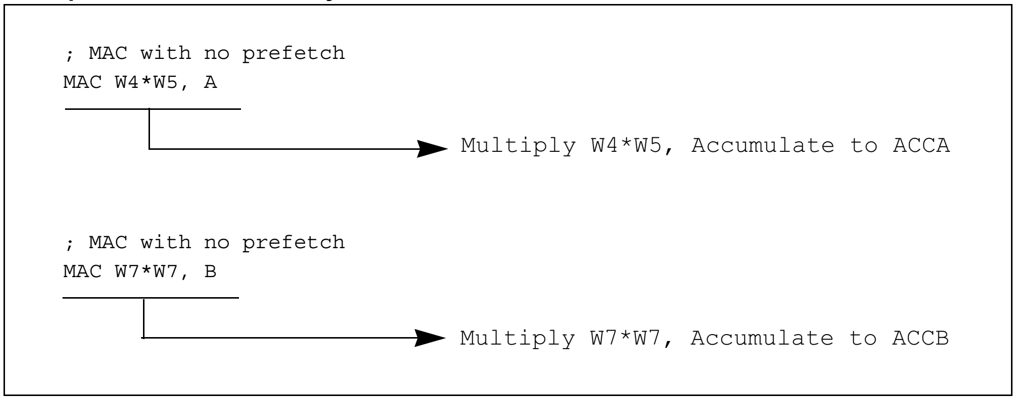
The write back ability of the MAC class of DSP instructions facilitates efficient processing of algorithms. This feature allows one mathematical operation to be performed with one accumulator, and the rounded contents of the other accumulator to be stored in the same cycle. As indicated in Table 4-9, register W13 is assigned for performing the write back, and two Addressing modes are supported: Direct and Indirect with Post-increment.

The CLR, MOVSAC and MSC instructions support accumulator write back, while the ED, EDAC, MPY and MPY.N instructions do not support accumulator write back. The MAC instruction, which multiplies two working registers which are not the same, also supports accumulator write back. However, the square and accumulate MAC instruction does not support accumulator write back (see Table 4.14).

4.14.5 MAC Syntax

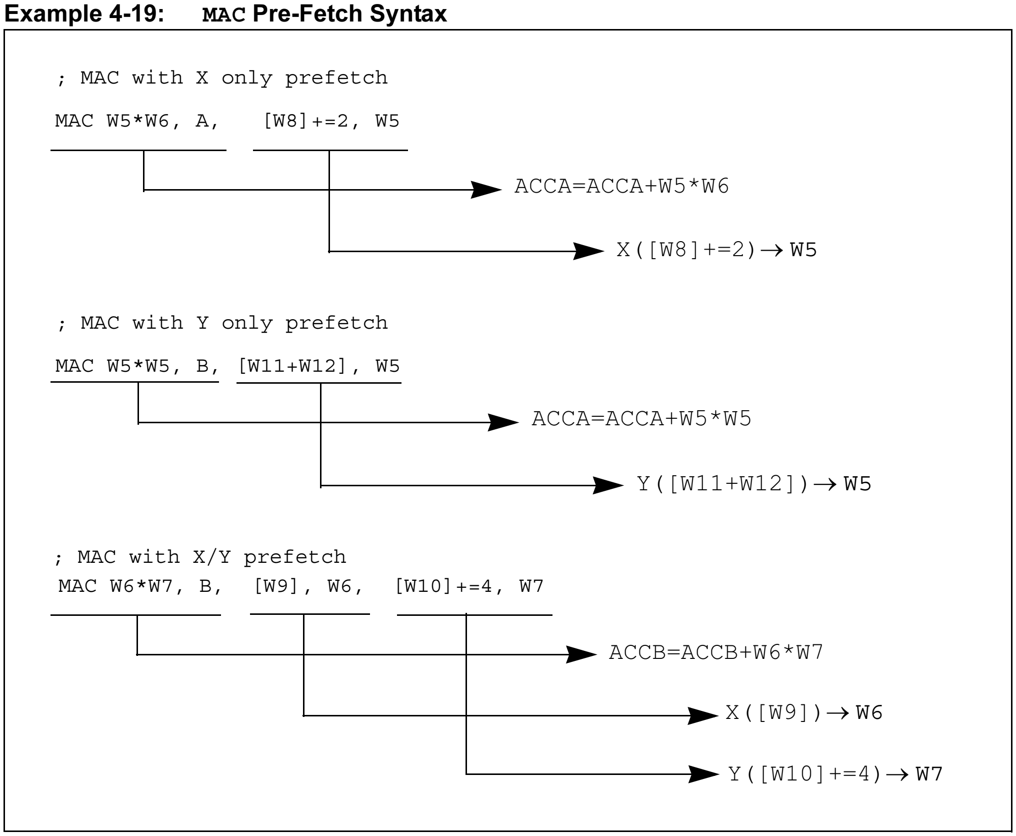
The syntax of the MAC class of instructions can have several formats, which depend on the instruction type and the operation it is performing, with respect to pre-fetches and accumulator write back. With the exception of the CLR and MOVSAC instructions, all MAC class instructions must specify a target accumulator along with two multiplicands, as shown in Example 4-18.

Example 4-18: Base MAC Syntax



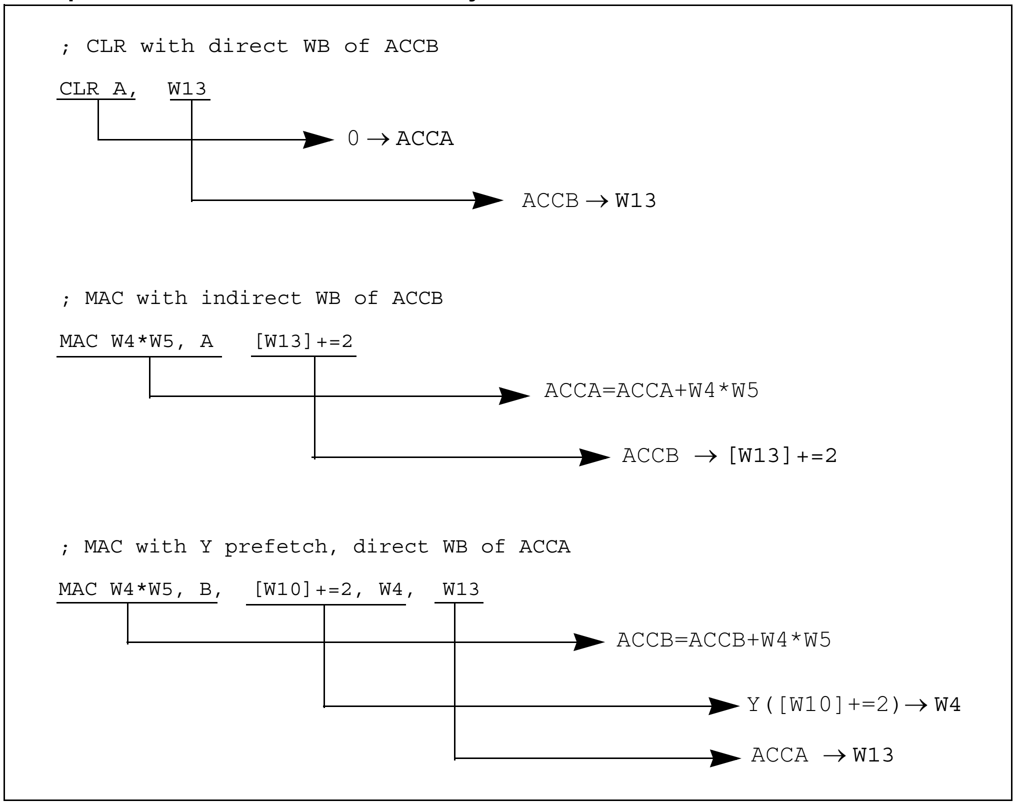
If a pre-fetch is used in the instruction, the assembler is capable of discriminating the X or Y data pre-fetch based on the register used for the effective address. [W8] or [W9] specifies the X pre-fetch and [W10] or [W11] specifies the Y pre-fetch. Brackets around the working register are required in the syntax, and they designate that indirect addressing is used to perform the pre-fetch. When address modification is used, it must be specified using a minus-equals or plus-equals “C”- like syntax (i.e., “[W8]-=2” or “[W8]+=6”). When Register Offset Addressing is used for the pre-fetch, W12 is placed inside the brackets ([W9+W12] for X pre-fetches and [W11+W12] for Y pre-fetches). Each pre-fetch operation must also specify a pre-fetch destination register (W4-W7). In the instruction syntax, the destination register appears before the pre-fetch register. Legal forms of pre-fetch are shown in Example 4-19.

Example 4-19: MAC Pre-Fetch Syntax



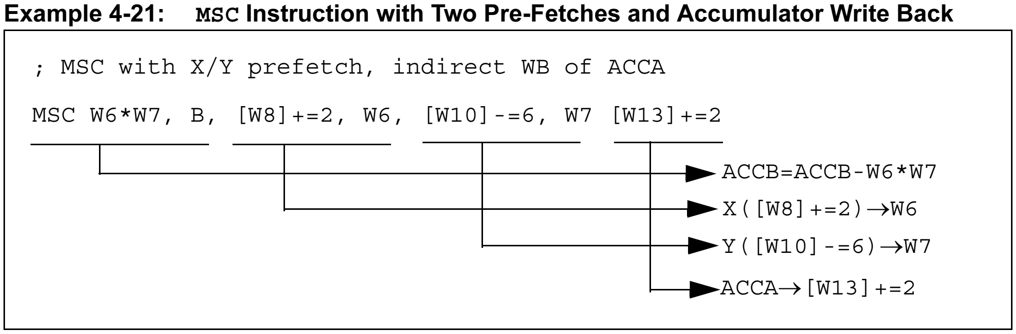
If an accumulator write back is used in the instruction, it is specified last. The write back must use the W13 register, and allowable forms for the write back are "W13" for direct addressing and "[W13]+=2" for indirect addressing with post-increment. By definition, the accumulator not used in the mathematical operation is stored, so the write back accumulator is **not** specified in the instruction. Legal forms of accumulator write back (WB) are shown in Example 4-20.

Example 4-20: `MAC` Accumulator WB Syntax



Putting it all together, an `MSC` instruction which performs two pre-fetches and a write back is shown in Example 4-21.

Example 4-21: `MSC` Instruction with Two Pre-Fetches and Accumulator Write Back



 Advance Information

4.15 DSP Accumulator Instructions

The DSP Accumulator instructions do not have pre-fetch or accumulator WB ability, but they do provide the ability to add, negate, shift, load and store the contents of either 40-bit accumulator. In addition, the ADD and SUB instructions allow the two accumulators to be added or subtracted from each other. DSP Accumulator instructions are shown in Table 4-12 and instruction details are provided in **Section 5. "Instruction Descriptions"**.

Table 4-12: DSP Accumulator Instructions

Instruction	Description	Accumulator WB?
ADD	Add accumulators	No
ADD	16-bit signed accumulator add	No
LAC	Load accumulator	No
NEG	Negate accumulator	No
SAC	Store accumulator	No
SAC.R	Store rounded accumulator	No
SFTAC	Arithmetic shift accumulator by Literal	No
SFTAC	Arithmetic shift accumulator by (Wn)	No
SUB	Subtract accumulators	No

4.16 Scaling Data with the FBCL Instruction

To minimize quantization errors that are associated with data processing using DSP instructions, it is important to utilize the complete numerical result of the operations. This may require scaling data up to avoid underflow (i.e., when processing data from a 12-bit ADC), or scaling data down to avoid overflow (i.e., when sending data to a 10-bit DAC). The scaling, which must be performed to minimize quantization error, depends on the dynamic range of the input data which is operated on, and the required dynamic range of the output data. At times, these conditions may be known beforehand and fixed scaling may be employed. In other cases, scaling conditions may not be fixed or known, and then dynamic scaling must be used to process data.

The FBCL instruction (Find First Bit Change Left) can efficiently be used to perform dynamic scaling, because it determines the exponent of a value. A fixed point or integer value's exponent represents the amount which the value may be shifted before overflowing. This information is valuable, because it may be used to bring the data value to "full scale", meaning that it's numeric representation utilizes all the bits of the register it is stored in.

The FBCL instruction determines the exponent of a word by detecting the first bit change starting from the value's sign bit and working towards the LSB. Since the dsPIC™ device's barrel shifter uses negative values to specify a left shift, the FBCL instruction returns the negated exponent of a value. If the value is being scaled up, this allows the ensuing shift to be performed immediately with the value returned by FBCL. Additionally, since the FBCL instruction only operates on signed quantities, FBCL produces results in the range of -15:0. When the FBCL instruction returns '0', it indicates that the value is already at full scale. When the instruction returns -15, it indicates that the value cannot be scaled (as is the case with 0x0 and 0xFFFF). Table 4-13 shows word data with various dynamic ranges, their exponents, and the value after scaling each data to maximize the dynamic range. Example 4-22 shows how the FBCL instruction may be used for block processing.

Table 4-13: Scaling Examples

Word Value	Exponent	Full Scale Value (Word Value << Exponent)
0x0001	14	0x4000
0x0002	13	0x4000
0x0004	12	0x4000
0x0100	6	0x4000
0x01FF	6	0x7FC0
0x0806	3	0x4030
0x2007	1	0x400E
0x4800	0	0x4800
0x7000	0	0x7000
0x8000	0	0x8000
0x900A	0	0x900A
0xE001	2	0x8004
0xFF07	7	0x8380

Note: For the word values `0x0000` and `0xFFFF`, the `FBCL` instruction returns -15.

As a practical example, assume that block processing is performed on a sequence of data with very low dynamic range stored in 1.15 fractional format. To minimize quantization errors, the data may be scaled up to prevent any quantization loss which may occur as it is processed. The `FBCL` instruction can be executed on the sample with the largest magnitude to determine the optimal scaling value for processing the data. Note that scaling the data up is performed by left shifting the data. This is demonstrated with the code snippet below.

Example 4-22: Scaling with `FBCL`

```
    ; assume W0 contains the largest absolute value of the data block
    ; assume W4 points to the beginning of the data block
    ; assume the block of data contains BLOCK_SIZE words

    ; determine the exponent to use for scaling
    FBCL    W0, W2             ; store exponent in W2

    ; scale the entire data block before processing
    DO      #(BLOCK_SIZE-1), SCALE
    LAC     [W4], A            ; move the next data sample to ACCA
    SFTAC   A, W2              ; shift ACCA by W2 bits
SCALE:
    SAC     A, [W4++]          ; store scaled input (overwrite original)

    ; now process the data
    ; (processing block goes here)
```


4.17 Normalizing the Accumulator with the FBCL Instruction

The process of scaling a quantized value for its maximum dynamic range is known as normalization (the data in the third column in Table 4-13 contains normalized data). Accumulator normalization is a technique used to ensure that the accumulator is properly aligned before storing data from the accumulator, and the FBCL instruction facilitates this function.

The two 40-bit accumulators each have 8 guard bits from the AccU register, which expands the dynamic range of the accumulators from 1.31 to 9.31, when operating in Super Saturation mode (see **Section 4.11.1 "Integer and Fractional Data"**). However, even in Super Saturation mode, the Store Rounded Accumulator (SAC.R) instruction only stores 16-bit data (in 1.15 format) from AccH, as described in **Section 4.12 "Accumulator Usage"**. Under certain conditions, this may pose a problem.

Proper data alignment for storing the contents of the accumulator may be achieved by scaling the accumulator down if AccU is in use, or scaling the accumulator up if all of the AccH bits are not being used. To perform such scaling, the FBCL instruction must operate on the AccU byte and it must operate on the AccH word. If a shift is required, the ALU's 40-bit shifter is employed, using the SFTAC instruction to perform the scaling. Example 4-23 contains a code snippet for accumulator normalization.

Example 4-23: Normalizing with FBCL

```
; assume an operation in ACCA has just completed (SR intact)
; assume the processor is in super saturation mode
; assume ACCAH is defined to be the address of ACCAH (0x24)

    MOV     #ACCAH, W5          ; W5 points to ACCAH
    BRA     OA, FBCL_GUARD      ; if overflow we right shift
FBCL_HI:
    FBCL    [W5], W0            ; extract exponent for left shift
    BRA     SHIFT_ACC           ; branch to the shift
FBCL_GUARD:
    FBCL    [++W5], W0          ; extract exponent for right shift
    ADD.B   W0, #15, W0         ; adjust the sign for right shift
SHIFT_ACC:
    SFTAC   A, W0               ; shift ACCA to normalize
```

4

Instruction Set Details

NOTES:

Section 5. Instruction Descriptions

HIGHLIGHTS

This section of the manual contains the following major topics:

5

Instruction Descriptions

5.1 Instruction Symbols

All symbols used in **Section 5.4 "Instruction Descriptions"** are shown in Table 1-2.

5.2 Instruction Encoding Field Descriptors Introduction

All instruction encoding field descriptors used in **Section 5.4 "Instruction Descriptions"** are shown in Table 5.2 through Table 5-12.

Table 5-1: Instruction Encoding Field Descriptors

Field	Description
A	Accumulator selection bit: 0=ACCA; 1=ACCB
aa	Accumulator Write Back mode (see Table 5-12)
B	Byte mode selection bit: 0=word operation; 1=byte operation
bbbb	4-bit bit position select: 0000=LSB; 1111=MSB
D	Destination address bit: 0=result stored in WREG; 1=result stored in file register
dddd	Wd destination register select: 0000=W0; 1111=W15
f ffff ffff ffff	13-bit register file address (0x0000 to 0x1FFF)
fff ffff ffff ffff	15-bit register file word address (implied 0 LSB) (0x0000 to 0xFFFE)
ffff ffff ffff ffff	16-bit register file byte address (0x0000 to 0xFFFF)
ggg	Register Offset Addressing mode for Ws source register (see Table 5-4)
hhh	Register Offset Addressing mode for Wd destination register (see Table 5-5)
iiii	Pre-Fetch X Operation (see Table 5-6)
jjjj	Pre-Fetch Y Operation (see Table 5-8)
k	1-bit literal field, constant data or expression
kkkk	4-bit literal field, constant data or expression
kk kkkk	6-bit literal field, constant data or expression
kkkk kkkk	8-bit literal field, constant data or expression
kk kkkk kkkk	10-bit literal field, constant data or expression
kk kkkk kkkk kkkk	14-bit literal field, constant data or expression
kkkk kkkk kkkk kkkk	16-bit literal field, constant data or expression
mm	Multiplier source select with same working registers (see Table 5-10)
mmm	Multiplier source select with different working registers (see Table 5-11)
nnnn nnnn nnnn nnn0 nnn nnnn	23-bit program address for CALL and GOTO instructions
nnnn nnnn nnnn nnnn	16-bit program offset field for relative branch/call instructions
ppp	Addressing mode for Ws source register (see Table 5-2)
qqq	Addressing mode for Wd destination register (see Table 5-3)
rrrr	Barrel shift count
ssss	Ws source register select: 0000=W0; 1111=W15
tttt	Dividend select, Most Significant Word
vvvv	Dividend select, Least Significant Word
W	Double-Word mode selection bit: 0=word operation; 1=double-word operation
wwww	Wb base register select: 0000=W0; 1111=W15
xx	Pre-Fetch X Destination (see Table 5-7)
xxxx xxxx xxxx xxxx	16-bit unused field (don't care)
yy	Pre-Fetch Y Destination (see Table 5-9)
z	Bit test destination: 0=C flag bit; 1=Z flag bit

Table 5-2: Addressing Modes for Ws Source Register

ppp	Addressing Mode	Source Operand
000	Register Direct	Ws
001	Indirect	[Ws]
010	Indirect with Post-Decrement	[Ws--]
011	Indirect with Post-Increment	[Ws++]
100	Indirect with Pre-Decrement	[--Ws]
101	Indirect with Pre-Increment	[++Ws]
11x	Unused	

Table 5-3: Addressing Modes for Wd Destination Register

qqq	Addressing Mode	Destination Operand
000	Register Direct	Wd
001	Indirect	[Wd]
010	Indirect with Post-Decrement	[Wd--]
011	Indirect with Post-Increment	[Wd++]
100	Indirect with Pre-Decrement	[--Wd]
101	Indirect with Pre-Increment	[++Wd]
11x	Unused (an attempt to use this Addressing mode will force a `RESET` instruction)	

Table 5-4: Offset Addressing Modes for Ws Source Register (with Register Offset)

ggg	Addressing Mode	Source Operand
000	Register Direct	Ws
001	Indirect	[Ws]
010	Indirect with Post-Decrement	[Ws--]
011	Indirect with Post-Increment	[Ws++]
100	Indirect with Pre-Decrement	[--Ws]
101	Indirect with Pre-Increment	[++Ws]
11x	Indirect with Register Offset	[Ws+Wb]

Table 5-5: Offset Addressing Modes for Wd Destination Register (with Register Offset)

hhh	Addressing Mode	Source Operand
000	Register Direct	Wd
001	Indirect	[Wd]
010	Indirect with Post-Decrement	[Wd--]
011	Indirect with Post-Increment	[Wd++]
100	Indirect with Pre-Decrement	[--Wd]
101	Indirect with Pre-Increment	[++Wd]
11x	Indirect with Register Offset	[Wd+Wb]

Table 5-6: X Data Space Pre-Fetch Operation

iiii	Operation
0000	Wxd=[W8]
0001	Wxd=[W8], W8 = W8 + 2
0010	Wxd=[W8], W8 = W8 + 4
0011	Wxd=[W8], W8 = W8 + 6
0100	No Pre-fetch for X Data Space
0101	Wxd=[W8], W8 = W8 – 6
0110	Wxd=[W8], W8 = W8 – 4
0111	Wxd=[W8], W8 = W8 – 2
1000	Wxd=[W9]
1001	Wxd=[W9], W9 = W9 + 2
1010	Wxd=[W9], W9 = W9 + 4
1011	Wxd=[W9], W9 = W9 + 6
1100	Wxd=[W9+W12]
1101	Wxd=[W9], W9 = W9 – 6
1110	Wxd=[W9], W9 = W9 – 4
1111	Wxd=[W9], W9 = W9 – 2

Table 5-7: X Data Space Pre-Fetch Destination

xx	Wxd
00	W4
01	W5
10	W6
11	W7

Table 5-8: Y Data Space Pre-Fetch Operation

jjjj	Operation
0000	Wyd=[W10]
0001	Wyd=[W10], W10 = W10 + 2
0010	Wyd=[W10], W10 = W10 + 4
0011	Wyd=[W10], W10 = W10 + 6
0100	No Pre-fetch for Y Data Space
0101	Wyd=[W10], W10 = W10 – 6
0110	Wyd=[W10], W10 = W10 – 4
0111	Wyd=[W10], W10 = W10 – 2
1000	Wyd=[W11]
1001	Wyd=[W11], W11 = W11 + 2
1010	Wyd=[W11], W11 = W11 + 4
1011	Wyd=[W11], W11 = W11 + 6
1100	Wyd=[W11+W12]
1101	Wyd=[W11], W11 = W11 – 6
1110	Wyd=[W11], W11 = W11 – 4
1111	Wyd=[W11], W11 = W11 – 2

Table 5-9: Y Data Space Pre-Fetch Destination

yy	Wyd
00	W4
01	W5
10	W6
11	W7

Table 5-10: MAC or MPY Source Operands (Same Working Register)

mm	Multiplicands
00	W4 * W4
01	W5 * W5
10	W6 * W6
11	W7 * W7

Table 5-11: MAC or MPY Source Operands (Different Working Register)

mmm	Multiplicands
000	W4 * W5
001	W4 * W6
010	W4 * W7
011	Invalid
100	W5 * W6
101	W5 * W7
110	W6 * W7
111	Invalid

Table 5-12: MAC Accumulator Write Back Selection

aa	Write Back Selection
00	W13 = Other Accumulator (Direct Addressing)
01	[W13]+=2 = Other Accumulator (Indirect Addressing with Post-Increment)
10	No Write Back
11	Invalid

5.3 Instruction Description Example

The example description below is for the fictitious instruction `FOO`. The following example instruction was created to demonstrate how the table fields (syntax, operands, operation, etc.) are used to describe the instructions presented in **Section 5.4 "Instruction Descriptions"**.

FOO

The Header field summarizes what the instruction does

Syntax:	*The Syntax field consists of an optional label, the instruction mnemonic, any optional extensions which exist for the instruction, and the operands for the instruction. Most instructions support more than one operand variant to support the various dsPIC30F Addressing modes. In these circumstances, all possible instruction operands are listed beneath each other (as in the case of op2a, op2b and op2c above). Optional operands are enclosed in braces.*
Operands:	*The Operands field describes the set of values which each of the operands may take. Operands may be accumulator registers, file registers, literal constants (signed or unsigned), or working registers.*
Operation:	*The Operation field summarizes the operation performed by the instruction.*
Status Affected:	*The Status Affected field describes which bits of the Status Register are affected by the instruction. Status bits are listed by bit position in descending order.*
Encoding:	*The Encoding field shows how the instruction is bit encoded. Individual bit fields are explained in the Description field, and complete encoding details are provided in Table 5.2.*
Description:	*The Description field describes in detail the operation performed by the instruction. A key for the encoding bits is also provided.*
Words:	*The Words field contains the number of program words that are used to store the instruction in memory.*
Cycles:	*The Cycles field contains the number of instruction cycles that are required to execute the instruction.*
Examples:	*The Examples field contains examples which demonstrate how the instruction operates. "Before" and "After" register snapshots are provided, which allow the user to clearly understand what operation the instruction performs.*

5.4 Instruction Descriptions

ADD

Add f to WREG

Syntax:	{label:}	ADD{.B}	f	{,WREG}		
Operands:	f ∈ [0 ... 8191]					
Operation:	(f) + (WREG) → destination designated by D					
Status Affected:	DC, N, OV, Z, C					
Encoding:	1011	0100	0BDf	ffff	ffff	ffff

Description: Add the contents of the default working register WREG to the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1 `ADD.B RAM100 ; Add WREG to RAM100 (Byte mode)`

	Before Instruction		After Instruction	
WREG	CC80	WREG	CC80	
RAM100	FFC0	RAM100	FF40	
SR	0000	SR	0005	(OV, C=1)

Example 2 `ADD RAM200, WREG ; Add RAM200 to WREG (Word mode)`

	Before Instruction		After Instruction	
WREG	CC80	WREG	CC40	
RAM200	FFC0	RAM200	FFC0	
SR	0000	SR	0001	(C=1)

ADD — Add Literal to Wn

Syntax:	{label:} ADD{.B} #lit10, Wn
Operands:	lit10 ∈ [0 ... 255] for byte operation lit10 ∈ [0 ... 1023] for word operation Wn ∈ [W0 ... W15]
Operation:	lit10 + (Wn) → Wn
Status Affected:	DC, N, OV, Z, C

Encoding:

1011	0000	0Bkk	kkkk	kkkk	dddd

Description: Add the 10-bit unsigned literal operand to the contents of the working register Wn and place the result back into the working register Wn.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'k' bits specify the literal operand.
The 'd' bits select the address of the working register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See **Section 4.6 "Using 10-bit Literal Operands"** for information on using 10-bit literal operands in Byte mode.

Words: 1

Cycles: 1

Example 1

```
ADD.B    #0xFF, W7    ; Add -1 to W7 (Byte mode)
```

	Before Instruction	After Instruction	
W7	12C0	12BF	
SR	0000	0009	(N,C=1)

Example 2

```
ADD      #0xFF, W1    ; Add 255 to W1 (Word mode)
```

	Before Instruction	After Instruction
W1	12C0	13BF
SR	0000	0000

ADD

Add Wb to Short Literal

Syntax:	{label:}	ADD{.B}	Wb,	#lit5,	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[++Wd]
					[--Wd]

Operands: Wb $\in$ [W0 ... W15]
lit5 $\in$ [0 ... 31]
Wd $\in$ [W0 ... W15]

Operation: (Wb) + lit5 $\rightarrow$ Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0100	0www	wBqq	qddd	d11k	kkkk

Description: Add the contents of the base register Wb to the 5-bit unsigned short literal operand and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (0 for word, 1 for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘k’ bits provide the literal operand, a five-bit integer number.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
ADD.B     W0, #0x1F, W7     ; Add W0 and 31 (Byte mode)
                            ; Store the result in W7
```

	Before Instruction		After Instruction	
W0	2290	W0	2290	
W7	12C0	W7	12AF	
SR	0000	SR	0008	(N=1)

Example 2

```
ADD       W3, #0x6, [--W4]  ; Add W3 and 6 (Word mode)
                            ; Store the result in [--W4]
```

	Before Instruction		After Instruction
W3	6006	W3	6006
W4	1000	W4	0FFE
Data 0FFE	DDEE	Data 0FFE	600C
Data 1000	DDEE	Data 1000	DDEE
SR	0000	SR	0000

ADD

Add Wb to Ws

Syntax:	{label:}	ADD{.B}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]
				[++Ws],	[++Wd]
				[--Ws],	[--Wd]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Wb) + (Ws) → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

`0100`	`0www`	`wBqq`	`qddd`	`dppp`	`ssss`

Description: Add the contents of the source register Ws and the contents of the base register Wb and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
ADD.B     W5, W6, W7      ; Add W5 to W6, store result in W7
                          ; (Byte mode)
```

	Before Instruction		After Instruction
W5	AB00	W5	AB00
W6	0030	W6	0030
W7	FFFF	W7	FF30
SR	0000	SR	0000

Example 2

```
ADD       W5, W6, W7      ; Add W5 to W6, store result in W7
                          ; (Word mode)
```

	Before Instruction		After Instruction
W5	AB00	W5	AB00
W6	0030	W6	0030
W7	FFFF	W7	AB30
SR	0000	SR	0008 (N=1)

ADD — Add Accumulators

Syntax:	{label:} ADD Acc
Operands:	Acc ∈ [A,B]
Operation:	If (Acc = A): (ACCA) + (ACCB) → ACCA Else: (ACCA) + (ACCB) → ACCB
Status Affected:	OA, OB, OAB, SA, SB, SAB
Encoding:	1100 1011 A000 0000 0000 0000
Description:	Add the contents of Accumulator A to the contents of Accumulator B and place the result in the selected accumulator. This instruction performs a 40-bit addition. The ‘A’ bit specifies the destination accumulator.
Words:	1
Cycles:	1

Example 1

```
ADD     A          ; Add ACCB to ACCA
```

	Before Instruction	After Instruction
ACCA	00 0022 3300	00 1855 7858
ACCB	00 1833 4558	00 1833 4558
SR	0000	0000

Example 2

```
ADD     B          ; Add ACCA to ACCB
                   ; Assume Super Saturation mode enabled
                   ; (ACCSAT=1, SATA=1, SATB=1)
```

	Before Instruction	After Instruction
ACCA	00 E111 2222	00 E111 2222
ACCB	00 7654 3210	01 5765 5432
SR	0000	4800 (OB, OAB=1)

ADD

16-Bit Signed Add to Accumulator

Syntax:	{label:}	ADD	Ws,	{#Slit4,}	Acc
			[Ws],		
			[Ws++],		
			[Ws--],		
			[--Ws],		
			[++Ws],		
			[Ws+Wb],		

Operands: Ws ∈ [W0 ... W15]
Wb ∈ [W0 ... W15]
Slit4 ∈ [-8 ... +7]
Acc ∈ [A,B]

Operation: $\text{Shift}_{\text{Slit4}}(\text{Extend(Ws)}) + (\text{Acc}) \rightarrow \text{Acc}$

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1100	1001	Awww	wrrr	rggg	ssss

Description: Add a 16-bit value specified by the source working register to the Most Significant word of the selected accumulator. The source operand may specify the direct contents of a working register or an effective address. The value specified is added to the Most Significant Word of the accumulator, by sign-extending and zero backfilling the source operand prior to the operation. The value added to the accumulator may also be shifted by a 4-bit signed literal before the addition is made.

The 'A' bit specifies the destination accumulator.
The 'w' bits specify the offset register Wb.
The 'r' bits encode the optional shift.
The 'g' bits select the source Address mode.
The 's' bits specify the source register Ws.

Note: Positive values of operand Slit4 represent an arithmetic shift right and negative values of operand Slit4 represent an arithmetic shift left. The contents of the source register are not affected by Slit4.

Words: 1

Cycles: 1

Example 1

```
ADD  W0, #2, A       ; Add W0 right-shifted by 2 to ACCA
```

	Before Instruction	After Instruction
W0	8000	8000
ACCA	00 7000 0000	00 5000 0000
SR	0000	0000

Example 2

```
ADD  [W5++], A      ; Add the effective value of W5 to ACCA
                    ; Post-increment W5
```

	Before Instruction		After Instruction
W5	2000	W5	2002
ACCA	00 0067 2345	ACCA	00 5067 2345
Data 2000	5000	Data 2000	5000
SR	0000	SR	0000

ADDC — Add f to WREG with Carry

Syntax:	{label:}	ADDC{.B}	f	{,WREG}		
Operands:	f ∈ [0 ... 8191]					
Operation:	(f) + (WREG) + (C) → destination designated by D					
Status Affected:	DC, N, OV, Z, C					
Encoding:	1011	0100	1BDf	ffff	ffff	ffff

Description: Add the contents of the default working register WREG, the contents of the file register and the Carry bit and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The WREG is set to working register W0.
3: The Z flag is "sticky" for `ADDC`, `CPB`, `SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
ADDC.B    RAM100        ; Add WREG and C bit to RAM100
                        ; (Byte mode)
```

	Before Instruction			After Instruction
WREG	CC60		WREG	CC60
RAM100	8006		RAM100	8067
SR	0001	(C=1)	SR	0000

Example 2

```
ADDC      RAM200, WREG  ; Add RAM200 and C bit to the WREG
                        ; (Word mode)
```

	Before Instruction			After Instruction	
WREG	5600		WREG	8A01	
RAM200	3400		RAM200	3400	
SR	0001	(C=1)	SR	000C	(N, OV=1)

ADDC

Add Literal to Wn with Carry

Syntax:	{label:}	ADDC{.B}	#lit10,	Wn	

Operands: lit10 ∈ [0 ... 255] for byte operation
lit10 ∈ [0 ... 1023] for word operation
Wn ∈ [W0 ... W15]

Operation: lit10 + (Wn) + (C) → Wn

Status Affected: DC, N, OV, Z, C

Encoding:

1011	0000	1Bkk	kkkk	kkkk	dddd

Description: Add the 10-bit unsigned literal operand, the contents of the working register Wn and the Carry bit and place the result back into the working register Wn.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘k’ bits specify the literal operand.
The ‘d’ bits select the address of the working register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See **Section 2.7 “Using 10-bit Literal Operands”** for information on using 10-bit literal operands in Byte mode.

3: The Z flag is “sticky” for `ADDC`, `CPB`, `SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
ADDC.B    #0xFF, W7       ; Add -1 and C bit to W7 (Byte mode)
```

	Before Instruction		After Instruction	
W7	12C0		12BF	
SR	0000	(C=0)	0009	(N,C=1)

Example 2

```
ADDC      #0xFF, W1       ; Add 255 and C bit to W1 (Word mode)
```

	Before Instruction		After Instruction
W1	12C0		13C0
SR	0001	(C=1)	0000

5

Instruction Descriptions

ADDC

Add Wb to Short Literal with Carry

Syntax:	{label:}	ADDC{.B}	Wb,	#lit5,	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[++Wd]
					[--Wd]

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
Wd ∈ [W0 ... W15]

Operation: (Wb) + lit5 + (C) → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0100	1www	wBqq	qddd	d11k	kkkk

Description: Add the contents of the base register Wb, the 5-bit unsigned short literal operand and the Carry bit and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'k' bits provide the literal operand, a five-bit integer number.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.
2: The Z flag is "sticky" for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
ADDC.B    W0, #0x1F, [W7]  ; Add W0, 31 and C bit (Byte mode)
                           ; Store the result in [W7]
```

	Before Instruction			After Instruction	
W0	CC80		W0	CC80	
W7	12C0		W7	12C0	
Data 12C0	B000		Data 12C0	B09F	
SR	0000	(C=0)	SR	0008	(N=1)

Example 2

```
ADDC      W3, #0x6, [--W4] ; Add W3, 6 and C bit (Word mode)
                           ; Store the result in [--W4]
```

	Before Instruction			After Instruction
W3	6006		W3	6006
W4	1000		W4	0FFE
Data 0FFE	DDEE		Data 0FFE	600D
Data 1000	DDEE		Data 1000	DDEE
SR	0001	(C=1)	SR	0000

ADDC

Add Wb to Ws with Carry

Syntax:	{label:}	ADDC{.B}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]
				[++Ws],	[++Wd]
				[--Ws],	[--Wd]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Wb) + (Ws) + (C) → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0100	1www	wBqq	qddd	dppp	ssss

Description: Add the contents of the source register Ws, the contents of the base register Wb and the Carry bit and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The Z flag is “sticky” for `ADDC`, `CPB`, `SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
ADDC.B    W0,[W1++],[W2++]    ; Add W0, [W1] and C bit (Byte mode)
                              ; Store the result in [W2]
                              ; Post-increment W1, W2
```

	Before Instruction			After Instruction
W0	CC20		W0	CC20
W1	0800		W1	0801
W2	1000		W2	1001
Data 0800	AB25		Data 0800	AB25
Data 1000	FFFF		Data 1000	FF46
SR	0001	(C=1)	SR	0000

5

Instruction Descriptions

Example 2

```
ADDC      W3,[W2++],[W1++]    ; Add W3, [W2] and C bit (Word mode)
                              ; Store the result in [W1]
                              ; Post-increment W1, W2
```

	Before Instruction			After Instruction
W1	1000		W1	1002
W2	2000		W2	2002
W3	0180		W3	0180
Data 1000	8000		Data 1000	2681
Data 2000	2500		Data 2000	2500
SR	0001	(C=1)	SR	0000

AND

AND f and WREG

Syntax:	{label:}	AND{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (f).AND.(WREG) → destination designated by D

Status Affected: N, Z

Encoding:

1011	0110	0BDf	ffff	ffff	ffff

Description: Compute the logical AND operation of the contents of the default working register WREG and the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
AND.B  RAM100          ; AND WREG to RAM100 (Byte mode)
```

	Before Instruction	After Instruction	
WREG	CC80	CC80	
RAM100	FFC0	FF80	
SR	0000	0008	(N=1)

Example 2

```
AND  RAM200, WREG      ; AND RAM200 to WREG (Word mode)
```

	Before Instruction	After Instruction
WREG	CC80	0080
RAM200	12C0	12C0
SR	0000	0000

AND — AND Literal and Wd

Syntax:	{label:}	AND{.B}	#lit10,	Wn

Operands: lit10 ∈ [0 ... 255] for byte operation
lit10 ∈ [0 ... 1023] for word operation
Wn ∈ [W0 ... W15]

Operation: lit10.AND.(Wn) → Wn

Status Affected: N, Z

Encoding:

1011	0010	0Bkk	kkkk	kkkk	dddd

Description: Compute the logical AND operation of the 10-bit literal operand and the contents of the working register Wn and place the result back into the working register Wn. Register direct addressing must be used for Wn.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘k’ bits specify the literal operand.
The ‘d’ bits select the address of the working register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See **Section 4.6 “Using 10-bit Literal Operands”** for information on using 10-bit literal operands in Byte mode.

Words: 1

Cycles: 1

Example 1

```
AND.B  #0x83, W7        ; AND 0x83 to W7 (Byte mode)
```

	Before Instruction	After Instruction	
W7	12C0	1280	
SR	0000	0008	(N=1)

Example 2

```
AND  #0x333, W1         ; AND 0x333 to W1 (Word mode)
```

	Before Instruction	After Instruction
W1	12D0	0210
SR	0000	0000

AND

AND Wb and Short Literal

Syntax:	{label:}	AND{.B}	Wb,	#lit5,	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[++Wd]
					[--Wd]

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
Wd ∈ [W0 ... W15]

Operation: (Wb).AND.lit5 → Wd

Status Affected: N, Z

Encoding:

0110	0www	wBqq	qddd	d11k	kkkk

Description: Compute the logical AND operation of the contents of the base register Wb and the 5-bit literal and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘k’ bits provide the literal operand, a five-bit integer number.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
AND.B    W0,#0x3,[W1++]  ; AND W0 and 0x3 (Byte mode)
                         ; Store to [W1]
                         ; Post-increment W1
```

	Before Instruction	After Instruction
W0	23A5	23A5
W1	2211	2212
Data 2210	9999	0199
SR	0000	0000

Example 2

```
AND      W0,#0x1F,W1     ; AND W0 and 0x1F (Word mode)
                         ; Store to W1
```

	Before Instruction	After Instruction
W0	6723	6723
W1	7878	0003
SR	0000	0000

AND

And Wb and Ws

Syntax:	{label:}	AND{.B}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]
				[++Ws],	[++Wd]
				[--Ws],	[--Wd]

Operands:
Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Wb).AND.(Ws) → Wd

Status Affected: N, Z

Encoding:

0110	0www	wBqq	qddd	dppp	ssss

Description: Compute the logical AND operation of the contents of the source register Ws and the contents of the base register Wb and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
AND.B     W0, W1 [W2++]  ; AND W0 and W1, and
                         ; store to [W2] (Byte mode)
                         ; Post-increment W2
```

	Before Instruction	After Instruction
W0	AA55	AA55
W1	2211	2211
W2	1001	1002
Data 1000	FFFF	11FF
SR	0000	0000

Example 2

```
AND     W0, [W1++], W2  ; AND W0 and [W1], and
                        ; store to W2 (Word mode)
                        ; Post-increment W1
```

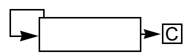
	Before Instruction		After Instruction
W0	AA55	W0	AA55
W1	1000	W1	1002
W2	55AA	W2	2214
Data 1000	2634	Data 1000	2634
SR	0000	SR	0000

ASR — Arithmetic Shift Right f

Syntax:	{label:}	ASR{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: For byte operation:
(f<7>) → Dest<7>
(f<7>) → Dest<6>
(f<6:1>) → Dest<5:0>
(f<0>) → C
For word operation:
(f<15>) → Dest<15>
(f<15>) → Dest<14>
(f<14:1>) → Dest<13:0>
(f<0>) → C



Status Affected: N, Z, C

Encoding:

1101	0101	1BDf	ffff	ffff	ffff

Description: Shift the contents of the file register one bit to the right and place the result in the destination register. The Least Significant bit of the file register is shifted into the Carry bit of the Status Register. After the shift is performed, the result is sign-extended. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘D’ bit selects the destination (`0` for WREG, `1` for file register).
The ‘f’ bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
ASR.B  RAM400, WREG     ; ASR RAM400 and store to WREG
                        ; (Byte mode)
```

	Before Instruction		After Instruction	
WREG	0600	WREG	0611	
RAM400	0823	RAM400	0823	
SR	0000	SR	0001	(C=1)

Example 2

```
ASR   RAM200            ; ASR RAM200 (Word mode)
```

	Before Instruction		After Instruction	
RAM200	8009	RAM200	C004	
SR	0000	SR	0009	(N, C=1)

 Advance Information

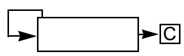
ASR

Arithmetic Shift Right Ws

Syntax:	{label:}	ASR{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: For byte operation:
(Ws<7>) → Wd<7>
(Ws<7>) → Wd<6>
(Ws<6:1>) → Wd<5:0>
(Ws<0>) → C
For word operation:
(Ws<15>) → Wd<15>
(Ws<15>) → Wd<14>
(Ws<14:1>) → Wd<13:0>
(Ws<0>) → C



Status Affected: N, Z, C

Encoding:

1101	0001	1Bqq	qddd	dppp	ssss

Description: Shift the contents of the source register Ws one bit to the right and place the result in the destination register Wd. The Least Significant bit of Ws is shifted into the Carry bit of the Status Register. After the shift is performed, the result is sign-extended. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
ASR.B  [W0++], [W1++]    ; ASR [W0] and store to [W1] (Byte mode)
                         ; Post-increment W0 and W1
```

	Before Instruction	After Instruction
W0	0600	0601
W1	0801	0802
Data 600	2366	2366
Data 800	FFC0	33C0
SR	0000	0000

Example 2

```
ASR  W12, W13            ; ASR W12 and store to W13 (Word mode)
```

	Before Instruction	After Instruction	
W12	AB01	AB01	
W13	0322	D580	
SR	0000	0009	(N, C=1)

ASR

Arithmetic Shift Right by Short Literal

Syntax:	{label:}	ASR	Wb,	#lit4,	Wnd

Operands: Wb ∈ [W0 ... W15]
lit4 ∈ [0...15]
Wnd ∈ [W0 ... W15]

Operation: lit4<3:0> → Shift_Val
Wb<15> → Wnd<15:15-Shift_Val+1>
Wb<15:Shift_Val> → Wnd<15-Shift_Val:0>

Status Affected: N, Z

Encoding:

1101	1110	1www	wddd	d100	kkkk

Description: Arithmetic shift right the contents of the source register Wb by the 4-bit unsigned literal and store the result in the destination register Wnd. After the shift is performed, the result is sign-extended. Direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the destination register.
The 'k' bits provide the literal operand.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: 1

Example 1

```
ASR  W0, #0x4, W1        ; ASR W0 by 4 and store to W1
```

	Before Instruction		After Instruction
W0	060F	W0	060F
W1	1234	W1	0060
SR	0000	SR	0000

Example 2

```
ASR  W0, #0x6, W1        ; ASR W0 by 6 and store to W1
```

	Before Instruction		After Instruction
W0	80FF	W0	80FF
W1	0060	W1	FE03
SR	0000	SR	0008 (N=1)

Example 3

```
ASR  W0, #0xF, W1        ; ASR W0 by 15 and store to W1
```

	Before Instruction		After Instruction
W0	70FF	W0	70FF
W1	CC26	W1	0000
SR	0000	SR	0002 (Z=1)

ASR

Arithmetic Shift Right by Wns

Syntax:	{label:}	ASR	Wb,	Wns,	Wnd

Operands: Wb ∈ [W0 ... W15]
Wns ∈ [W0 ...W15]
Wnd ∈ [W0 ... W15]

Operation: Wns<3:0> → Shift_Val
Wb<15> → Wnd<15:15-Shift_Val+1>
Wb<15:Shift_Val> → Wnd<15-Shift_Val:0>

Status Affected: N, Z

Encoding:

1101	1110	1www	wddd	d000	ssss

Description: Arithmetic shift right the contents of the source register Wb by the 4 Least Significant bits of Wns (up to 15 positions) and store the result in the destination register Wnd. After the shift is performed, the result is sign-extended. Direct addressing must be used for Wb, Wns and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the destination register.
The 's' bits select the address of the source register.

Note 1: This instruction operates in Word mode only.
2: If Wns is greater than 15, Wnd = 0x0 if Wb is positive, and Wnd = `0xFFFF` if Wb is negative.

Words: 1

Cycles: 1

Example 1

```
ASR  W0, W5, W6          ; ASR W0 by W5 and store to W6
```

	Before Instruction		After Instruction
W0	80FF	W0	80FF
W5	0004	W5	0004
W6	2633	W6	F80F
SR	0000	SR	0000

Example 2

```
ASR  W0, W5, W6          ; ASR W0 by W5 and store to W6
```

	Before Instruction		After Instruction
W0	6688	W0	6688
W5	000A	W5	000A
W6	FF00	W6	0019
SR	0000	SR	0000

Example 3

```
ASR  W11, W12, W13       ; ASR W11 by W12 and store to W13
```

	Before Instruction		After Instruction
W11	8765	W11	8765
W12	88E4	W12	88E4
W13	A5A5	W13	F876
SR	0000	SR	0008 (N=1)

Advance Information

BCLR

Bit Clear f

Syntax:	{label:}	BCLR{.B}	f,	#bit4

Operands: f ∈ [0 ... 8191] for byte operation
f ∈ [0 ... 8190] (even only) for word operation
bit4 ∈ [0 ... 7] for byte operation
bit4 ∈ [0 ... 15] for byte operation

Operation: 0 → f<bit4>

Status Affected: None

Encoding:

1010	1001	bbbf	ffff	ffff	fffb

Description: Clear the bit in the file register f specified by 'bit4'. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations).

The 'b' bits select value bit4 of the bit position to be cleared.
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: When this instruction operates in Word mode, the file register address must be word aligned.
3: When this instruction operates in Byte mode, 'bit4' must be between 0 and 7.

Words: 1

Cycles: 1

Example 1

```
BCLR.B 0x800, #0x7      ; Clear bit 7 in 0x800
```

	Before Instruction		After Instruction
Data 0800	66EF	Data 0800	666F
SR	0000	SR	0000

Example 2

```
BCLR   0x400, #0xA      ; Clear bit 10 in 0x400
```

	Before Instruction		After Instruction
Data 0400	AA55	Data 0400	A855
SR	0000	SR	0000

BCLR — Bit Clear in Ws

Syntax:	{label:}	BCLR{.B}	Ws,	#bit4
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
bit4 ∈ [0 ... 7] for byte operation
bit4 ∈ [0 ... 15] for word operation

Operation: 0 → Ws<bit4>

Status Affected: None

Encoding:

1010	0001	bbbb	0B00	0ppp	ssss

Description: Clear the bit in register Ws specified by ‘bit4’. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations). Register direct or indirect addressing may be used for Ws.

The ‘b’ bits select value bit4 of the bit position to be cleared.
The ‘B’ bit selects byte or word operation (0 for word, 1 for byte).
The ‘s’ bits select the address of the source/destination register.
The ‘p’ bits select the source Address mode.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: When this instruction operates in Word mode, the source register address must be word aligned.
3: When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

Words: 1

Cycles: 1

Example 1

```
BCLR.B  W2, #0x2        ; Clear bit 3 in W2
```

	Before Instruction		After Instruction
W2	F234	W2	F230
SR	0000	SR	0000

Example 2

```
BCLR  [W0++], #0x0      ; Clear bit 0 in [W0]
                        ; Post-increment W0
```

	Before Instruction		After Instruction
W0	2300	W0	2302
Data 2300	5607	Data 2300	5606
SR	0000	SR	0000

BRA — Branch Unconditionally

Syntax:	{label:} BRA Expr
Operands:	Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].
Operation:	(PC+2) + 2*Slit16 → PC NOP → Instruction Register
Status Affected:	None
Encoding:	0011 0111 nnnn nnnn nnnn nnnn
Description:	The program will branch unconditionally, relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression. After the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The 'n' bits are a signed literal that specifies the number of program words offset from (PC+2).
Words:	1
Cycles:	2

Example 1

```
002000 HERE:    BRA THERE            ; Branch to THERE
002002          . . .
002004          . . .
002006          . . .
002008          . . .
00200A THERE:   . . .
00200C          . . .
```

	Before Instruction	After Instruction
PC	00 2000	00 200A
SR	0000	0000

Example 2

```
002000 HERE:    BRA THERE+0x2        ; Branch to THERE+0x2
002002          . . .
002004          . . .
002006          . . .
002008          . . .
00200A THERE:   . . .
00200C          . . .
```

	Before Instruction	After Instruction
PC	00 2000	00 200C
SR	0000	0000

Example 3

```
002000 HERE:    BRA 0x1366           ; Branch to 0x1366
002002          . . .
002004          . . .
```

	Before Instruction	After Instruction
PC	00 2000	00 1366
SR	0000	0000

BRA

Computed Branch

Syntax:	{label:}	BRA	Wn			
Operands:	Wn ∈ [W0 ... W15]					
Operation:	(PC+2) + (2*Wn) → PC NOP → Instruction Register					
Status Affected:	None					
Encoding:	0000	0001	0110	0000	0000	ssss
Description:	The program will branch unconditionally, relative to the next PC. The offset of the branch is the sign-extended 17-bit value (2*Wn), which supports branches up to 32K instructions forward or backward. After this instruction executes, the new PC will be (PC+2)+2*Wn, since the PC will have incremented to fetch the next instruction. The 's' bits select the address of the source register.					
Words:	1					
Cycles:	2					

Example 1

```
002000 HERE:    BRA W7              ; Branch forward 2*W7
002002          . . .
 ...            . . .
 ...            . . .
002106          . . .
002108 TABLE7:  . . .
00210A          . . .
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 2108
W7	0084	W7	0084
SR	0000	SR	0000

 Advance Information

BRA C

Branch if Carry

Syntax:	{label:}	BRA	C,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation: Condition = C
If (Condition)
(PC+2) + 2*Slit16 → PC
NOP → Instruction Register

Status Affected: None

Encoding:

0011	0001	nnnn	nnnn	nnnn	nnnn

Description: If the Carry flag bit is '1', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a 16-bit signed literal that specify the offset from (PC+2) in instruction words.

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA C, CARRY   ; If C is set, branch to CARRY
002002 NO_C:     . . .          ; Otherwise... continue
002004           . . .
002006           GOTO THERE
002008 CARRY:    . . .
00200A           . . .
00200C THERE:    . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 2008	
SR	0001	(C=1)	SR	0001	(C=1)

Example 2

```
002000 HERE:     BRA C, CARRY   ; If C is set, branch to CARRY
002002 NO_C:     ...            ; Otherwise... continue
002004           ...
002006           GOTO THERE
002008 CARRY:    ...
00200A           ...
00200C THERE:    ...
00200E           ...
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 2002
SR	0000	SR	0000

Example 3

```
006230 HERE:     BRA C, CARRY   ; If C is set, branch to CARRY
006232 NO_C:     ...            ; Otherwise... continue
006234           ...
006236           GOTO THERE
006238 CARRY:    ...
00623A           ...
00623C THERE:    ...
00623E           ...
```

	Before Instruction		After Instruction
PC	00 6230	PC	00 6238
SR	0001 (C=1)	SR	0001 (C=1)

Example 4

```
006230 START:    ...
006232           ...
006234 CARRY:    ...
006236           ...
006238           ...
00623A           ...
00623C HERE:     BRA C, CARRY   ; If C is set, branch to CARRY
00623E           ...            ; Otherwise... continue
```

	Before Instruction		After Instruction
PC	00 623C	PC	00 6234
SR	0001 (C=1)	SR	0001 (C=1)

BRA GE

Branch if Signed Greater Than or Equal

Syntax:	{label:}	BRA	GE,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = (N&&OV)||(!N&&!OV)
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0011	1101	nnnn	nnnn	nnnn	nnnn

Description: If the logical expression (N&&OV)||(!N&&!OV) is true, then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a `NOP` executed in the second cycle.

The 'n' bits are a 16-bit signed literal that specify the offset from (PC+2) in instruction words.

Note: The assembler will convert the specified label into the offset to be used.

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
007600 LOOP:    . . .
007602          . . .
007604          . . .
007606          . . .
007608 HERE:    BRA GE, LOOP       ; If GE, branch to LOOP
00760A NO_GE:   . . .              ; Otherwise... continue
```

	Before Instruction		After Instruction
PC	00 7608	PC	00 7600
SR	0000	SR	0000

Example 2

```
007600 LOOP:    . . .
007602          . . .
007604          . . .
007606          . . .
007608 HERE:    BRA GE, LOOP       ; If GE, branch to LOOP
00760A NO_GE:   . . .              ; Otherwise... continue
```

	Before Instruction		After Instruction
PC	00 7608	PC	00 760A
SR	0008 (N=1)	SR	0008 (N=1)

BRA GEU — Branch if Unsigned Greater Than or Equal

Syntax:	{label:}	BRA	GEU,	Expr		

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16 offset that supports an offset range of [-32768 ... +32767] program words.

Operation: Condition = C
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0011	0001	nnnn	nnnn	nnnn	nnnn

Description: If the Carry flag is '1', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a 16-bit signed literal that specify the offset from (PC+2) in instruction words.

Note: This instruction is identical to the BRA C, Expr (Branch if Carry) instruction and has the same encoding. It will reverse assemble as BRA C, Slit16.

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA GEU, BYPASS       ; If C is set, branch
002002 NO_GEU:   . . .                 ; to BYPASS
002004           . . .                 ; Otherwise... continue
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 200C	
SR	0001	(C=1)	SR	0001	(C=1)

BRA GT

Branch if Signed Greater Than

Syntax:	{label:}	BRA	GT,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = (!Z&&N&&OV)||(!Z&&!N&&!OV)
If (Condition)
(PC+2) + 2*Slit16 → PC
NOP → Instruction Register

Status Affected: None

Encoding:

0011	1100	nnnn	nnnn	nnnn	nnnn

Description: If the logical expression (!Z&&N&&OV)||(!Z&&!N&&!OV) is true, then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a `NOP` executed in the second cycle.

The 'n' bits are a 16-bit signed literal that specify the offset from (PC+2) in instruction words.

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA GT, BYPASS      ; If GT, branch to BYPASS
002002 NO_GT:    . . .               ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 200C	
SR	0001	(C=1)	SR	0001	(C=1)

BRA GTU — Branch if Unsigned Greater Than

Syntax:	{label:} BRA GTU, Expr
Operands:	Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].
Operation:	Condition = (C&&!Z) If (Condition) (PC+2) + 2*Slit16 → PC NOP → Instruction Register
Status Affected:	None
Encoding:	0011 | 1110 | nnnn | nnnn | nnnn | nnnn
Description:	If the logical expression (C&&!Z) is true, then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression. If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a `NOP` executed in the second cycle. The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).
Words:	1
Cycles:	1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA GTU, BYPASS    ; If GTU, branch to  BYPASS
002002 NO_GTU:   . . .              ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction		After Instruction	
PC	00 2000		00 200C	
SR	0001	(C=1)	0001	(C=1)

BRA LE — Branch if Signed Less Than or Equal

Syntax:	{label:}	BRA	LE,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = Z||(N&&!OV)||(!N&&OV)
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0011	0100	nnnn	nnnn	nnnn	nnnn

Description: If the logical expression (Z||(N&&!OV)||(!N&&OV)) is true, then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a `NOP` executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA LE, BYPASS      ; If LE, branch to
002002 NO_LE:   . . .                  BYPASS
002004           . . .               ; Otherwise... continue
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 2002	
SR	0001	(C=1)	SR	0001	(C=1)

BRA LEU — Branch if Unsigned Less Than or Equal

Syntax:	{label:}	BRA	LEU,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 $\in$ [-32768 ... +32767].

Operation: Condition = !C||Z
If (Condition)
  (PC+2) + 2*Slit16 $\rightarrow$ PC
  NOP $\rightarrow$ Instruction Register

Status Affected: None

Encoding:

0011	0110	nnnn	nnnn	nnnn	nnnn

Description: If the logical expression (!C||Z) is true, then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a `NOP` executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA LEU, BYPASS     ; If LEU, branch to BYPASS
002002 NO_LEU:   . . .               ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 200C	
SR	0001	(C=1)	SR	0001	(C=1)

 Advance Information

BRA LT

Branch if Signed Less Than

Syntax:	{label:}	BRA	LT,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 $\in$ [-32768 ... +32767].

Operation: Condition = (N&&!OV)||(!N&&OV)
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0011	0101	nnnn	nnnn	nnnn	nnnn

Description: If the logical expression ((N&&!OV)||(!N&&OV)) is true, then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a `NOP` executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA LT, BYPASS     ; If LT, branch to BYPASS
002002 NO_LT:    . . .              ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 2002	
SR	0001	(C=1)	SR	0001	(C=1)

BRA LTU

Branch if Unsigned Less Than

Syntax:	{label:} BRA LTU, Expr
Operands:	Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].
Operation:	Condition = !C If (Condition) (PC+2) + 2*Slit16 → PC NOP → Instruction Register
Status Affected:	None
Encoding:	0011 | 1001 | nnnn | nnnn | nnnn | nnnn
Description:	If the Carry flag is '0', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression. If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle. The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2). **Note:** This instruction is identical to the BRA NC, Expr (Branch if Not Carry) instruction and has the same encoding. It will reverse assemble as BRA NC, Slit16.
Words:	1
Cycles:	1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA LTU, BYPASS    ; If LTU, branch to BYPASS
002002 NO_LTU:   . . .              ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction		After Instruction	
PC	00 2000		00 2002	
SR	0001	(C=1)	0001	(C=1)

BRA N

Branch if Negative

Syntax:	{label:}	BRA	N,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = N
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register.

Status Affected: None

Encoding:

0011	0011	nnnn	nnnn	nnnn	nnnn

Description: If the Negative flag is ‘1’, then the program will branch relative to the next PC. The offset of the branch is the 2’s complement number ‘2*Slit16’, which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA N, BYPASS       ; If N, branch to BYPASS
002002 NO_N:     . . .               ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 200C	
SR	0008	(N=1)	SR	0008	(N=1)

BRA NC

Branch if Not Carry

Syntax:	{label:} BRA NC, Expr
Operands:	Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].
Operation:	Condition = !C If (Condition) (PC+2) + 2*Slit16 → PC NOP → Instruction Register
Status Affected:	None
Encoding:	`0011` `1001` `nnnn` `nnnn` `nnnn` `nnnn`
Description:	If the Carry flag is '0', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression. If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle. The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).
Words:	1
Cycles:	1 (2 if branch taken)

Example 1

```
002000 HERE:      BRA NC, BYPASS      ; If NC, branch to BYPASS
002002 NO_NC:     . . .               ; Otherwise... continue
002004            . . .
002006            . . .
002008            . . .
00200A            GOTO THERE
00200C BYPASS:    . . .
00200E            . . .
```

	Before Instruction		After Instruction	
PC	00 2000		00 2002	
SR	0001	(C=1)	0001	(C=1)

 Advance Information

BRA NN

Branch if Not Negative

Syntax:	{label:}	BRA	NN,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = !N
If (Condition)
(PC+2) + 2*Slit16 → PC
NOP → Instruction Register

Status Affected: None

Encoding:

0011	1011	nnnn	nnnn	nnnn	nnnn

Description: If the Negative flag is '0', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:      BRA NN, BYPASS     ; If NN, branch to BYPASS
002002 NO_NN:     . . .              ; Otherwise... continue
002004            . . .
002006            . . .
002008            . . .
00200A            GOTO THERE
00200C BYPASS:    . . .
00200E            . . .
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 200C
SR	0000	SR	0000

5

Instruction Descriptions

BRA NOV

Branch if Not Overflow

Syntax:	{label:}	BRA	NOV,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = !OV
If (Condition)
(PC+2) + 2*Slit16 → PC
NOP → Instruction Register

Status Affected: None

Encoding:

0011	1000	nnnn	nnnn	nnnn	nnnn

Description: If the Overflow flag is '0', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a `NOP` executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA NOV, BYPASS     ; If NOV, branch to BYPASS
002002 NO_NOV:   . . .               ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 200C	
SR	0008	(N=1)	SR	0008	(N=1)

BRA NZ

Branch if Not Zero

Syntax:	{label:}	BRA	NZ,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = !Z
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0011	1010	nnnn	nnnn	nnnn	nnnn

Description: If the Z flag is '0', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA NZ, BYPASS     ; If NZ, branch to BYPASS
002002 NO_NZ:    . . .              ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 2002	
SR	0002	(Z=1)	SR	0002	(Z=1)

BRA OA — Branch if Overflow Accumulator A

Syntax:	{label:}	BRA	OA,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = OA
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0000	1100	nnnn	nnnn	nnnn	nnnn

Description: If the Overflow Accumulator A flag is '1', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Note: The assembler will convert the specified label into the offset to be used.

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA OA, BYPASS      ; If OA, branch to BYPASS
002002 NO_OA:    . . .               ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 200C	
SR	8800	(OA, OAB=1)	SR	8800	(OA, OAB=1)

 Advance Information

BRA OB

Branch if Overflow Accumulator B

Syntax:	{label:}	BRA	OB,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = OB
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0000	1101	nnnn	nnnn	nnnn	nnnn

Description: If the Overflow Accumulator B flag is '1', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA OB, BYPASS      ; If OB, branch to BYPASS
002002 NO_OB:    . . .               ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 2002	
SR	8800	(OA, OAB=1)	SR	8800	(OA, OAB=1)

BRA OV — Branch if Overflow

Syntax:	{label:}	BRA	OV,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = OV
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0011	0000	nnnn	nnnn	nnnn	nnnn

Description: If the Overflow flag is '1', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a `NOP` executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA OV, BYPASS     ; If OV, branch to BYPASS
002002 NO_OV     . . .              ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 2002	
SR	0002	(Z=1)	SR	0002	(Z=1)

 Advance Information

BRA SA

Branch if Saturation Accumulator A

Syntax:	{label:}	BRA	SA,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = SA
If (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0000	1110	nnnn	nnnn	nnnn	nnnn

Description: If the Saturation Accumulator A flag is ‘1’, then the program will branch relative to the next PC. The offset of the branch is the 2’s complement number ‘2*Slit16’, which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA SA, BYPASS     ; If SA, branch to BYPASS
002002 NO_SA:    . . .              ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 200C	
SR	2400	(SA, SAB=1)	SR	2400	(SA, SAB=1)

BRA SB — Branch if Saturation Accumulator B

Syntax:	{label:}	BRA	SB,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = SB
if (Condition)
  (PC+2) + 2*Slit16→ PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0000	1111	nnnn	nnnn	nnnn	nnnn

Description: If the Saturation Accumulator B flag is '1', then the program will branch relative to the next PC. The offset of the branch is the 2's complement number '2*Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a `NOP` executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA SB, BYPASS    ; If SB, branch to BYPASS
002002 NO_SB:    . . .             ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction	After Instruction
PC	00 2000	00 2002
SR	0000	0000

 Advance Information

BRA Z — Branch if Zero

Syntax:	{label:}	BRA	Z,	Expr

Operands: Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ [-32768 ... +32767].

Operation: Condition = Z
if (Condition)
  (PC+2) + 2*Slit16 → PC
  NOP → Instruction Register

Status Affected: None

Encoding:

0011	0010	nnnn	nnnn	nnnn	nnnn

Description: If the Zero flag is ‘1’, then the program will branch relative to the next PC. The offset of the branch is the 2’s complement number ‘2*Slit16’, which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC+2) + 2*Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a signed literal that specifies the number of instructions offset from (PC+2).

Words: 1

Cycles: 1 (2 if branch taken)

Example 1

```
002000 HERE:     BRA Z, BYPASS       ; If Z, branch to BYPASS
002002 NO_Z:     . . .               ; Otherwise... continue
002004           . . .
002006           . . .
002008           . . .
00200A           GOTO THERE
00200C BYPASS:   . . .
00200E           . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 200C	
SR	0002	(Z=1)	SR	0002	(Z=1)

BSET

Bit Set f

Syntax:	{label:}	BSET{.B}	f,	#bit4

Operands: f ∈ [0 ... 8191] for byte operation
f ∈ [0 ... 8190] (even only) for word operation
bit4 ∈ [0 ... 7] for byte operation
bit4 ∈ [0 ... 15] for word operation

Operation: 1 → f<bit4>

Status Affected: None

Encoding:

1010	1000	bbbf	ffff	ffff	fffb

Description: Set the bit in the file register f specified by 'bit4'. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations).

The 'b' bits select value bit4 of the bit position to be set.
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: When this instruction operates in Word mode, the file register address must be word aligned.

3: When this instruction operates in Byte mode, 'bit4' must be between 0 and 7.

Words: 1

Cycles: 1

Example 1

```
BSET.B  0x601, #0x3      ; Set bit 3 in 0x601
```

	Before Instruction	After Instruction
Data 0600	F234	FA34
SR	0000	0000

Example 2

```
BSET    0x444, #0xF      ; Set bit 15 in 0x444
```

	Before Instruction	After Instruction
Data 0444	5604	D604
SR	0000	0000

BSET

Bit Set in Ws

Syntax:	{label:}	BSET{.B}	Ws,	#bit4
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
bit4 ∈ [0 ... 7] for byte operation
bit4 ∈ [0 ... 15] for word operation

Operation: 1 → Ws<bit4>

Status Affected: None

Encoding:

1010	0000	bbbb	0B00	0ppp	ssss

Description: Set the bit in register Ws specified by 'bit4'. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations). Register direct or indirect addressing may be used for Ws.

The 'b' bits select value bit4 of the bit position to be cleared.
The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'p' bits select the source Address mode.
The 's' bits select the address of the source/destination register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: When this instruction operates in Word mode, the source register address must be word aligned.
3: When this instruction operates in Byte mode, 'bit4' must be between 0 and 7.

Words: 1

Cycles: 1

Example 1

```
BSET.B  W3, #0x7        ; Set bit 7 in W3
```

	Before Instruction	After Instruction
W3	0026	00A6
SR	0000	0000

Example 2

```
BSET  [W4++], #0x0      ; Set bit 0 in [W4]
                        ; Post-increment W4
```

	Before Instruction	After Instruction
W4	6700	6702
Data 6700	1734	1735
SR	0000	0000

5

Instruction Descriptions

BSW

Bit Write in Ws

Syntax:	{label:}	BSW.C	Ws,	Wb
		BSW.Z	[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
Wb ∈ [W0 ... W15]

Operation: <u>For ".C" operation:</u>
C → Ws<(Wb)>
<u>For ".Z" operation (default):</u>
$\overline{Z}$ → Ws<(Wb)>

Status Affected: None

Encoding:

1010	1101	Zwww	w000	0ppp	ssss

Description: The (Wb) bit in register Ws is written with the value of the C or $\overline{Z}$ flag from the Status register. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the working register. Only the four Least Significant bits of Wb are used to determine the destination bit number. Register direct addressing must be used for Wb, and either register direct, or indirect addressing may be used for Ws.

The 'Z' bit selects the C or Z flag as source.
The 'w' bits select the address of the bit select register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: This instruction only operates in Word mode. If no extension is provided, the ".Z" operation is assumed.

Words: 1

Cycles: 1

Example 1

```
BSW.C  W2, W3              ; Set bit W3 in W2 to the value
                           ; of the C bit
```

	Before Instruction			After Instruction	
W2	F234		W2	7234	
W3	111F		W3	111F	
SR	0002	(Z=1, C=0)	SR	0002	(Z=1, C=0)

Example 2

```
BSW.Z  W2, W3              ; Set bit W3 in W2 to the complement
                           ; of the Z bit
```

	Before Instruction			After Instruction	
W2	E235		W2	E234	
W3	0550		W3	0550	
SR	0002	(Z=1, C=0)	SR	0002	(Z=1, C=0)

Example 3

```
BSW.C   [++W0], W6          ; Set bit W6 in [W0++] to the value
                            ; of the C bit
```

	Before Instruction			After Instruction	
W0	1000		W0	1002	
W6	34A3		W6	34A3	
Data 1002	2380		Data 1002	2388	
SR	0001	(Z=0, C=1)	SR	0001	(Z=0, C=1)

Example 4

```
BSW     [W1--], W5          ; Set bit W5 in [W1] to the
                            ; complement of the Z bit
                            ; Post-decrement W1
```

	Before Instruction			After Instruction	
W1	1000		W1	0FFE	
W5	888B		W5	888B	
Data 1000	C4DD		Data 1000	CCDD	
SR	0001	(C=1)	SR	0001	(C=1)

BTG

Bit Toggle f

Syntax:	{label:} BTG{.B} f, #bit4
Operands:	f ∈ [0 ... 8191] for byte operation f ∈ [0 ... 8190] (even only) for word operation bit4 ∈ [0 ... 7] for byte operation bit4 ∈ [0 ... 15] for word operation
Operation:	$\overline{\text{(f)<bit4>}} \rightarrow$ (f)<bit4>
Status Affected:	None
Encoding:	`1010` `1010` `bbbf` `ffff` `ffff` `fffb`

1010	1010	bbbf	ffff	ffff	fffb

Description: Bit ‘bit4’ in file register f is toggled (complemented). For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operation, bit 15 for word operation) of the byte.

The ‘b’ bits select value bit4, the bit position to toggle.
The ‘f’ bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: When this instruction operates in Word mode, the file register address must be word aligned.

3: When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

Words: 1

Cycles: 1

Example 1

```
BTG.B   0x1001, #0x4   ; Toggle bit 4 in 0x1001
```

	Before Instruction		After Instruction
Data 1000	F234	Data 1000	E234
SR	0000	SR	0000

Example 2

```
BTG     0x1660, #0x8   ; Toggle bit 8 in RAM660
```

	Before Instruction		After Instruction
Data 1660	5606	Data 1660	5706
SR	0000	SR	0000

BTG — Bit Toggle in Ws

Syntax:	{label:}	BTG{.B}	Ws,	#bit4
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
bit4 ∈ [0 ... 7] for byte operation
bit4 ∈ [0 ... 15] for word operation

Operation: $\overline{\text{(Ws)<bit4>}} \rightarrow$ Ws<bit4>

Status Affected: None

Encoding:

1010	0010	bbbb	0B00	0ppp	ssss

Description: Bit ‘bit4’ in register Ws is toggled (complemented). For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations). Register direct or indirect addressing may be used for Ws.

The ‘b’ bits select value bit4, the bit position to test.
The ‘B’ bit selects byte or word operation (0 for word, 1 for byte).
The ‘s’ bits select the address of the source/destination register.
The ‘p’ bits select the source Address mode.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: When this instruction operates in Word mode, the source register address must be word aligned.
3: When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

Words: 1

Cycles: 1

Example 1

```
BTG  W2, #0x0          ; Toggle bit 0 in W2
```

	Before Instruction	After Instruction
W2	F234	F235
SR	0000	0000

Example 2

```
BTG  [W0++], #0x0      ; Toggle bit 0 in [W0]
                       ; Post-increment W0
```

	Before Instruction	After Instruction
W0	2300	2302
Data 2300	5606	5607
SR	0000	0000

BTSC

Bit Test f, Skip if Clear

Syntax:	{label:}	BTSC{.B}	f,	#bit4		

Operands: f ∈ [0 ... 8191] for byte operation
f ∈ [0 ... 8190] (even only) for word operation
bit4 ∈ [0 ... 7] for byte operation
bit4 ∈ [0 ... 15] for word operation

Operation: Test (f)<bit4>, skip if clear

Status Affected: None

Encoding:

1010	1111	bbbf	ffff	ffff	fffb

Description: Bit 'bit4' in the file register is tested. If the tested bit is '0', the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a NOP is executed instead. If the tested bit is '1', the next instruction is executed as normal. In either case, the contents of the file register are not changed. For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations).

The 'b' bits select value bit4, the bit position to test.
The 'f' bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: When this instruction operates in Word mode, the file register address must be word aligned.

3: When this instruction operates in Byte mode, 'bit4' must be between 0 and 7.

Words: 1

Cycles: 1 (2 or 3)

Example 1

```
002000 HERE:   BTSC.B  0x1201, #2  ; If bit 2 of 0x1201 is 0,
002002         GOTO    BYPASS      ; skip the GOTO
002004         . . .
002006         . . .
002008 BYPASS: . . .
00200A         . . .
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 2002
Data 1200	264F	Data 1200	264F
SR	0000	SR	0000

Example 2

```
002000 HERE:    BTSC    0x804, #14 ; If bit 14 of 0x804 is 0,
002002          GOTO    BYPASS     ; skip the GOTO
002004          . . .
002006          . . .
002008 BYPASS:  . . .
00200A          . . .
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 2004
Data 0804	2647	Data 0804	2647
SR	0000	SR	0000

BTSC — Bit Test Ws, Skip if Clear

Syntax:	{label:}	BTSC	Ws,	#bit4
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
bit4 ∈ [0 ... 15]

Operation: Test (Ws)<bit4>, skip if clear

Status Affected: None

Encoding:

1010	0111	bbbb	0000	0ppp	ssss

Description: Bit 'bit4' in Ws is tested. If the tested bit is '0', the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a NOP is executed instead. If the tested bit is '1', the next instruction is executed as normal. In either case, the contents of Ws are not changed. For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the word. Either register direct or indirect addressing may be used for Ws.

The 'b' bits select value bit4, the bit position to test.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: 1 (2 or 3 if the next instruction is skipped)

Example 1

```
002000 HERE:    BTSC   W0, #0x0      ; If bit 0 of W0 is 0,
002002          GOTO   BYPASS        ; skip the GOTO
002004          . . .
002006          . . .
002008 BYPASS:  . . .
00200A          . . .
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 2002
W0	264F	W0	264F
SR	0000	SR	0000

Example 2

```
002000 HERE:    BTSC    W6, #0xF       ; If bit 15 of W6 is 0,
002002          GOTO    BYPASS         ; skip the GOTO
002004          . . .
002006          . . .
002008 BYPASS:  . . .
00200A          . . .
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 2004
W6	264F	W6	264F
SR	0000	SR	0000

Example 3

```
003400 HERE:    BTSC    [W6++], #0xC  ; If bit 12 of [W6] is 0,
003402          GOTO    BYPASS        ; skip the GOTO
003404          . . .                 ; Post-increment W6
003406          . . .
003408 BYPASS:  . . .
00340A          . . .
```

	Before Instruction		After Instruction
PC	00 3400	PC	00 3402
W6	1800	W6	1802
Data 1800	1000	Data 1800	1000
SR	0000	SR	0000

BTSS — Bit Test f, Skip if Set

Syntax:	{label:} BTSS{.B} f, #bit4
Operands:	f ∈ [0 ... 8191] for byte operation f ∈ [0 ... 8190] (even only) for word operation bit4 ∈ [0 ... 7] for byte operation bit4 ∈ [0 ... 15] for word operation
Operation:	Test (f)<bit4>, skip if set
Status Affected:	None

Encoding:

1010	1110	bbbf	ffff	ffff	fffb

Description: Bit 'bit4' in the file register f is tested. If the tested bit is '1', the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a `NOP` is executed instead. If the tested bit is '0', the next instruction is executed as normal. In either case, the contents of the file register are not changed. For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operation, bit 15 for word operation).

The 'b' bits select value bit4, the bit position to test.
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: When this instruction operates in Word mode, the file register address must be word aligned.
3: When this instruction operates in Byte mode, 'bit4' must be between 0 and 7.

Words: 1

Cycles: 1 (2 or 3 if the next instruction is skipped)

Example 1

```
007100 HERE:     BTSS.B   0x1401, #0x1 ; If bit 1 of 0x1401 is 1,
007102           CLR      WREG         ; don't clear WREG
007104           . . .
```

	Before Instruction		After Instruction
PC	00 7100	PC	00 7104
Data 1400	0280	Data 1400	0280
SR	0000	SR	0000

Example 2

```
007100 HERE:     BTSS     0x890, #0x9  ; If bit 9 of 0x890 is 1,
007102           GOTO     BYPASS       ; skip the GOTO
007104           . . .
007106 BYPASS:   . . .
```

	Before Instruction		After Instruction
PC	00 7100	PC	00 7102
Data 0890	00FE	Data 0890	00FE
SR	0000	SR	0000

BTSS — Bit Test Ws, Skip if Set

Syntax:	{label:}	BTSS	Ws,	#bit4
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
bit4 ∈ [0 ... 15]

Operation: Test (Ws)<bit4>, skip if set.

Status Affected: None

Encoding:

1010	0110	bbbb	0000	0ppp	ssss

Description: Bit 'bit4' in Ws is tested. If the tested bit is '1', the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a `NOP` is executed instead. If the tested bit is '0', the next instruction is executed as normal. In either case, the contents of Ws are not changed. For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the word. Either register direct or indirect addressing may be used for Ws.

The 'b' bits select the value bit4, the bit position to test.
The 's' bits select the address of the source register.
The 'p' bits select the source Address mode.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: 1 (2 or 3 if the next instruction is skipped)

Example 1

```
002000 HERE:    BTSS    W0, #0x0       ; If bit 0 of W0 is 1,
002002          GOTO    BYPASS         ; skip the GOTO
002004          . . .
002006          . . .
002008 BYPASS:  . . .
00200A          . . .
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 2004
W0	264F	W0	264F
SR	0000	SR	0000

Example 2

```
002000 HERE:    BTSS    W6, #0xF      ; If bit 15 of W6 is 1,
002002          GOTO    BYPASS        ; skip the GOTO
002004          . . .
002006          . . .
002008 BYPASS:  . . .
00200A          . . .
```

	Before Instruction	After Instruction
PC	00 2000	00 2002
W6	264F	264F
SR	0000	0000

Example 3

```
003400 HERE:    BTSS    [W6++], 0xC   ; If bit 12 of [W6] is 1,
003402          GOTO    BYPASS        ; skip the GOTO
003404          . . .                 ; Post-increment W6
003406          . . .
003408 BYPASS:  . . .
00340A          . . .
```

	Before Instruction	After Instruction
PC	00 3400	00 3404
W6	1800	1802
Data 1800	1000	1000
SR	0000	0000

 Advance Information

BTST   Bit Test f

Syntax:	{label:}	BTST{.B}	f,	#bit4		
Operands:	f ∈ [0 ... 8191] for byte operation f ∈ [0 ... 8190] (even only) for word operation bit4 ∈ [0 ... 7] for byte operation bit4 ∈ [0 ... 15] for word operation					
Operation:	$\overline{\text{(f)<bit4>}} \rightarrow$ Z					
Status Affected:	Z					
Encoding:	1010	1011	bbbf	ffff	ffff	fffb

Description: Bit ‘bit4’ in file register f is tested and the complement of the tested bit is stored to the Z flag in the Status Register. The contents of the file register are not changed. For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operation, bit 15 for word operation).

The ‘b’ bits select value bit4, the bit position to be tested.
The ‘f’ bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: When this instruction operates in Word mode, the file register address must be word aligned.
3: When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

Words: 1

Cycles: 1

Example 1
```
BTST.B  0x1201, #0x3     ; Set Z = complement of
                         ; bit 3 in 0x1201
```

	Before Instruction		After Instruction	
Data 1200	F7FF	Data 1200	F7FF	
SR	0000	SR	0002	(Z=1)

Example 2
```
BTST    0x1302, #0x7     ; Set Z = complement of
                         ; bit 7 in 0x1302
```

	Before Instruction			After Instruction
Data 1302	F7FF		Data 1302	F7FF
SR	0002	(Z=1)	SR	0000

BTST — Bit Test in Ws

Syntax:	{label:}	BTST.C	Ws,	#bit4
		BTST.Z	[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
bit4 ∈ [0 ... 15]

Operation:
For ".C" operation:
$(Ws)<bit4> \rightarrow C$
For ".Z" operation (default):
$\overline{(Ws)<bit4>} \rightarrow Z$

Status Affected: Z or C

Encoding:

1010	0011	bbbb	Z000	0ppp	ssss

Description: Bit 'bit4' in register Ws is tested. If the "`.Z`" option of the instruction is specified, the complement of the tested bit is stored to the Zero flag in the Status register. If the "`.C`" option of the instruction is specified, the value of the tested bit is stored to the Carry flag in the Status register. In either case, the contents of Ws are not changed.

For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the word. Either register direct or indirect addressing may be used for Ws.
The 'b' bits select value bit4, the bit position to test.
The 'Z' bit selects the C or Z flag as destination.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: This instruction only operates in Word mode. If no extension is provided, the "`.Z`" operation is assumed.

Words: 1

Cycles: 1

Example 1

```
BTST.C  [W0++], #0x3     ; Set C = bit 3 in [W0]
                         ; Post-increment W0
```

	Before Instruction		After Instruction
W0	1200		1202
Data 1200	FFF7		FFF7
SR	0001 (C=1)		0000

Example 2

```
BTST.Z  W0, #0x7         ; Set Z = complement of bit 7 in W0
```

	Before Instruction		After Instruction
W0	F234		F234
SR	0000		0002 (Z=1)

BTST

Bit Test in Ws

Syntax:	{label:}	BTST.C	Ws,	Wb
		BTST.Z	[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
Wb ∈ [W0 ... W15]

Operation: For ".C" operation:
(Ws)<(Wb)> → C
For ".Z" operation (default):
$\overline{\text{(Ws)<(Wb)>}}$ → Z

Status Affected: Z or C

Encoding:

1010	0101	Zwww	w000	0ppp	ssss

Description: The (Wb) bit in register Ws is tested. If the "`.C`" option of the instruction is specified, the value of the tested bit is stored to the Carry flag in the Status register. If the "`.Z`" option of the instruction is specified, the complement of the tested bit is stored to the Zero flag in the Status register. In either case, the contents of Ws are not changed.

Only the four Least Significant bits of Wb are used to determine the bit number. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the working register. Register direct or indirect addressing may be used for Ws.

The 'Z' bit selects the C or Z flag as destination.
The 'w' bits select the address of the bit select register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: This instruction only operates in Word mode. If no extension is provided, the "`.Z`" operation is assumed.

Words: 1

Cycles: 1

Example 1

```
BTST.C  W2, W3          ; Set C = bit W3 of W2
```

	Before Instruction			After Instruction
W2	F234		W2	F234
W3	2368		W3	2368
SR	0001	(C=1)	SR	0000

5

Instruction Descriptions

Example 2

```
BTST.Z  [W0++], W1        ; Set Z = complement of
                          ; bit W1 in [W0],
                          ; Post-increment W0
```

	Before Instruction			After Instruction
W0	1200		W0	1202
W1	CCC0		W1	CCC0
Data 1200	6243		Data 1200	6243
SR	0002	(Z=1)	SR	0000

 Advance Information

BTSTS — Bit Test/Set f

Syntax:	{label:} BTSTS{.B} f, #bit4
Operands:	f ∈ [0 ... 8191] for byte operation f ∈ [0 ... 8190] (even only) for word operation bit4 ∈ [0 ... 7] for byte operation bit4 ∈ [0 ... 15] for word operation
Operation:	$\overline{\text{(f)<bit4>}} \rightarrow \text{Z}$ $1 \rightarrow \text{(f)<bit4>}$
Status Affected:	Z

Encoding:

1010	1100	bbbf	ffff	ffff	fffb

Description: Bit ‘bit4’ in file register f is tested and the complement of the tested bit is stored to the Zero flag in the Status register. The tested bit is then set to “1” in the file register. For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations).

The ‘b’ bits select value bit4, the bit position to test/set.
The ‘f’ bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: When this instruction operates in Word mode, the file register address must be word aligned.
3: When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

Words: 1

Cycles: 1

Example 1

```
BTSTS.B  0x1201, #0x3 ; Set Z = complement of bit 3 in 0x1201,
                      ; then set bit 3 of 0x1201 = 1
```

	Before Instruction		After Instruction	
Data 1200	F7FF	Data 1200	FFFF	
SR	0000	SR	0002	(Z=1)

Example 2

```
BTSTS    0x808, #15   ; Set Z = complement of bit 15 in 0x808,
                      ; then set bit 15 of 0x808 = 1
```

	Before Instruction			After Instruction
RAM300	8050		RAM300	8050
SR	0002	(Z=1)	SR	0000

BTSTS

Bit Test/Set in Ws

Syntax:	{label:}	BTSTS.C	Ws,	#bit4
		BTSTS.Z	[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: $Ws \in$ [W0 ... W15]
bit4 $\in$ [0 ... 15]

Operation: For ".C" operation:
(Ws)<bit4> $\rightarrow$ C
1 $\rightarrow$ Ws<bit4>
For ".Z" operation (default):
$\overline{\text{(Ws)<bit4>}} \rightarrow$ Z
1 $\rightarrow$ Ws<bit4>

Status Affected: Z or C

Encoding:

1010	0100	bbbb	Z000	0ppp	ssss

Description: Bit 'bit4' in register Ws is tested. If the "`.Z`" option of the instruction is specified, the complement of the tested bit is stored to the Zero flag in the Status register. If the "`.C`" option of the instruction is specified, the value of the tested bit is stored to the Carry flag in the Status register. In both cases, the tested bit in Ws is set to "`1`".

The 'b' bits select the value bit4, the bit position to test/set.
The 'Z' bit selects the C or Z flag as destination.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: This instruction only operates in Word mode. If no extension is provided, the "`.Z`" operation is assumed.

Words: 1

Cycles: 1

Example 1

```
BTSTS.C  [W0++], #0x3     ; Set C = bit 3 in [W0]
                          ; Set bit 3 in [W0] = 1
                          ; Post-increment W0
```

	Before Instruction		After Instruction
W0	1200	W0	1202
Data 1200	FFF7	Data 1200	FFFF
SR	0001 (C=1)	SR	0000

Example 2

```
BTSTS.Z  W0, #0x7         ; Set Z = complement of bit 7
                          ; in W0, and set bit 7 in W0 = 1
```

	Before Instruction		After Instruction
W0	F234	W0	F2BC
SR	0000	SR	0002 (Z=1)

CALL

Call Subroutine

Syntax:	{label:}	CALL	Expr			

Operands: Expr may be a label or expression (but not a literal).
Expr is resolved by the linker to a lit23, where lit23 $\in$ [0 ... 8388606].

Operation:
(PC)+4 $\rightarrow$ PC
(PC<15:0>) $\rightarrow$ (TOS)
(W15)+2 $\rightarrow$ W15
(PC<23:16>) $\rightarrow$ (TOS)
(W15)+2 $\rightarrow$ W15
lit23 $\rightarrow$ PC
NOP $\rightarrow$ Instruction Register

Status Affected: None

Encoding:

1st word	0000	0010	nnnn	nnnn	nnnn	nnn0
2nd word	0000	0000	0000	0000	0nnn	nnnn

Description: Direct subroutine call over the entire 4 Mbyte instruction program memory range. Before the call is made, the 24-bit return address (PC+4) is pushed onto the stack. After the return address is stacked, the 23-bit value 'lit23' is loaded into the PC.

The 'n' bits form the target address.

Note: The linker will resolve the specified expression into the lit23 to be used.

Words: 2

Cycles: 2

Example 1

```
026000          CALL   _FIR             ; Call _FIR subroutine
026004          MOV    W0, W1
   .            ...
   .            ...
026844 _FIR:    MOV   #0x400, W2        ; _FIR subroutine start
026846          ...
```

	Before Instruction		After Instruction
PC	02 6000	PC	02 6844
W15	A268	W15	A26C
Data A268	FFFF	Data A268	6004
Data A26A	FFFF	Data A26A	0002
SR	0000	SR	0000

Example 2

```
072000          CALL    _G66         ; call routine _G66
072004          MOV     W0, W1
   .            ...
077A28 _G66:    INC     W6, [W7++]   ; routine start
077A2A          ...
077A2C
```

	Before Instruction		After Instruction
PC	07 2000	PC	07 7A28
W15	9004	W15	9008
Data 9004	FFFF	Data 9004	2004
Data 9006	FFFF	Data 9006	0007
SR	0000	SR	0000

CALL — Call Indirect Subroutine

Syntax:	{label:} CALL Wn
Operands:	Wn ∈ [W0 ... W15]
Operation:	(PC)+2 → PC (PC<15:0>) → TOS (W15)+2 → W15 (PC<23:16>) → TOS (W15)+2 → W15 0 → PC<22:16> (Wn<15:1>) → PC<15:1> NOP → Instruction Register
Status Affected:	None

Encoding:

0000	0001	0000	0000	0000	ssss

Description:	Indirect subroutine call over the first 32K instructions of program memory. Before the call is made, the 24-bit return address (PC+2) is pushed onto the stack. After the return address is stacked, Wn<15:1> is loaded into PC<15:1> and PC<22:16> is cleared. Since PC<0> is always '0', Wn<0> is ignored. The 's' bits select the address of the source register.
Words:	1
Cycles:	2

Example 1

```
001002          CALL   W0          ; Call BOOT subroutine indirectly
001004          ...                ; using W0
   .            ...
001600 _BOOT:   MOV #0x400, W2     ; _BOOT starts here
001602          MOV #0x300, W6
   .            ...
```

	Before Instruction		After Instruction
PC	00 1002	PC	00 1600
W0	1600	W0	1600
W15	6F00	W15	6F04
Data 6F00	FFFF	Data 6F00	1004
Data 6F02	FFFF	Data 6F02	0000
SR	0000	SR	0000

Example 2

```
004200          CALL   W7          ; Call TEST subroutine indirectly
004202          ...                ; using W7
   .            ...
005500 _TEST:   INC    W1, W2      ; _TEST starts here
005502          DEC    W1, W3      ;
   .            ...
```

	Before Instruction		After Instruction
PC	00 4200	PC	00 5500
W7	5500	W7	5500
W15	6F00	W15	6F04
Data 6F00	FFFF	Data 6F00	4202
Data 6F02	FFFF	Data 6F02	0000
SR	0000	SR	0000

CLR

Clear f or WREG

Syntax:	{label:}	CLR{.B}	f WREG

Operands: f ∈ [0 ... 8191]

Operation: 0 → destination designated by D

Status Affected: None

Encoding:

1110	1111	0BDf	ffff	ffff	ffff

Description: Clear the contents of a file register or the default working register WREG. If WREG is specified, the WREG is cleared. Otherwise, the specified file register f is cleared.

The ‘B’ bit selects byte or word operation (0 for word, 1 for byte).
The ‘D’ bit selects the destination (0 for WREG, 1 for file register).
The ‘f’ bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
CLR.B  RAM200        ; Clear RAM200 (Byte mode)
```

	Before Instruction		After Instruction
RAM200	8009	RAM200	8000
SR	0000	SR	0000

Example 2

```
CLR    WREG          ; Clear WREG (Word mode)
```

	Before Instruction		After Instruction
WREG	0600	WREG	0000
SR	0000	SR	0000

CLR — Clear Wd

Syntax:	{label:}	CLR{.B}	Wd
			[Wd]
			[Wd++]
			[Wd--]
			[++Wd]
			[--Wd]

Operands: Wd ∈ [W0 ... W15]

Operation: 0 → Wd

Status Affected: None

Encoding:

1110	1011	0Bqq	qddd	d000	0000

Description: Clear the contents of register Wd. Either register direct or indirect addressing may be used for Wd.

The 'B' bits selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
CLR.B  W2          ; Clear W2 (Byte mode)
```

	Before Instruction	After Instruction
W2	3333	3300
SR	0000	0000

Example 2

```
CLR    [W0++]      ; Clear [W0]
                   ; Post-increment W0
```

	Before Instruction	After Instruction
W0	2300	2302
Data 2300	5607	0000
SR	0000	0000

CLR

Clear Accumulator, Pre-Fetch Operands

Syntax:	{label:}	CLR	Acc	{,[Wx],Wxd}	{,[Wy],Wyd}	{,AWB}
				{,[Wx]+=kx,Wxd}	{,[Wy]+=ky,Wyd}	
				{,[Wx]-=kx,Wxd}	{,[Wy]-=ky,Wyd}	
				{,[W9+W12],Wxd}	{,[W11+W12],Wyd}	

Operands: Acc ∈ [A,B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]
AWB ∈ [W13, [W13]+=2]

Operation: 0 → Acc(A or B)
([Wx])→ Wxd; (Wx)+/-kx→Wx
([Wy])→ Wyd; (Wy)+/-ky→Wy
(Acc(B or A)) rounded → AWB

Status Affected: OA, OB, SA, SB

Encoding:

1100	0011	A0xx	yyii	iijj	jjaa

Description: Clear all 40 bits of the specified accumulator, optionally pre-fetch operands in preparation for a MAC type instruction and optionally store the non-specified accumulator results. This instruction clears the respective overflow and saturate flags (either OA, SA or OB, SB).

Operands Wx, Wxd, Wy and Wyd specify optional pre-fetch operations which support indirect and register offset addressing, as described in **Section 4.14.1 "MAC Pre-Fetches"**. Operand AWB specifies the optional register direct or indirect store of the convergently rounded contents of the "other" accumulator, as described in **Section 4.14.4 "MAC Write Back"**.

The 'A' bit selects the other accumulator used for write back.
The 'x' bits select the pre-fetch Wxd destination.
The 'y' bits select the pre-fetch Wyd destination.
The 'i' bits select the Wx pre-fetch operation.
The 'j' bits select the Wy pre-fetch operation.
The 'a' bits select the accumulator write back destination.

Words: 1

Cycles: 1

Example 1

```
CLR  A, [W8]+=2, W4, W13   ; Clear ACCA
                           ; Load W4 with [W8], post-inc W8
                           ; Store ACCB to W13
```

	Before Instruction	After Instruction
W4	F001	1221
W8	2000	2002
W13	C623	5420
ACCA	00 0067 2345	00 0000 0000
ACCB	00 5420 3BDD	00 5420 3BDD
Data 2000	1221	1221
SR	0000	0000

Example 2

```
CLR   B, [W8]+=2, W6, [W10]+=2, W7, [W13]+=2 ; Clear ACCB
                                              ; Load W6 with [W8]
                                              ; Load W7 with [W10]
                                              ; Save ACCA to [W13]
                                              ; Post-inc W8,W10,W13
```

	Before Instruction	After Instruction
W6	F001	1221
W7	C783	FF80
W8	2000	2002
W10	3000	3002
W13	4000	4002
ACCA	00 0067 2345	00 0067 2345
ACCB	00 5420 ABDD	00 0000 0000
Data 2000	1221	1221
Data 3000	FF80	FF80
Data 4000	FFC3	0067
SR	0000	0000

CLRWDT — Clear Watchdog Timer

Syntax:	{label:}	CLRWDT
Operands:	None	
Operation:	0 → WDT count register 0 → WDT prescaler A count 0 → WDT prescaler B count	
Status Affected:	None	

Encoding:

1111	1110	0110	0000	0000	0000

Description:	Clear the contents of the Watchdog Timer count register and the prescaler count registers. The Watchdog Prescaler A and Prescaler B settings, set by configuration fuses in the FWDT, are not changed.
Words:	1
Cycles:	1

Example 1

```
CLRWDT     ; Clear Watchdog Timer
```

	Before Instruction	After Instruction
SR	0000	0000

COM — Complement f

Syntax:	{label:}	COM{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: $\overline{(f)} \rightarrow$ destination designated by D

Status Affected: N, Z

Encoding:

1110	1110	1BDf	ffff	ffff	ffff

Description: Compute the 1's complement of the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
COM.b  RAM200    ; COM RAM200 (Byte mode)
```

	Before Instruction	After Instruction	
RAM200	80FF	8000	
SR	0000	0002	(Z)

Example 2

```
COM    RAM400, WREG     ; COM RAM400 and store to WREG
                        ; (Word mode)
```

	Before Instruction	After Instruction	
WREG	1211	F7DC	
RAM400	0823	0823	
SR	0000	0008	(N=1)

COM

Complement Ws

Syntax:	{label:}	COM{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: $Ws \in$ [W0 ... W15]
$Wd \in$ [W0 ... W15]

Operation: $\overline{(Ws)} \rightarrow Wd$

Status Affected: N, Z

Encoding:

1110	1010	1Bqq	qddd	dppp	ssss

Description: Compute the 1's complement of the contents of the source register Ws and place the result in the destination register Wd. Either register direct or indirect addressing may be used for both Ws and Wd.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
COM.B  [W0++], [W1++]   ; COM [W0] and store to [W1] (Byte mode)
                        ; Post-increment W0, W1
```

	Before Instruction	After Instruction	
W0	2301	2302	
W1	2400	2401	
Data 2300	5607	5607	
Data 2400	ABCD	ABA9	
SR	0000	0008	(N=1)

Example 2

```
COM   W0, [W1++]     ; COM W0 and store to [W1] (Word mode)
                     ; Post-increment W1
```

	Before Instruction	After Instruction
W0	D004	D004
W1	1000	1002
Data 1000	ABA9	2FFB
SR	0000	0000

CP

Compare f with WREG, Set Status Flags

Syntax:	{label:} CP{.B} f					
Operands:	f ∈ [0 ...8191]					
Operation:	(f) – (WREG)					
Status Affected:	DC, N, OV, Z, C					
Encoding:	1110	0011	0B0f	ffff	ffff	ffff

Description: Compute (f) – (WREG) and update the Status register. This instruction is equivalent to the SUBWF instruction, but the result of the subtraction is not stored.

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'f' bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
CP.B    RAM400    ; Compare RAM400 with WREG (Byte mode)
```

	Before Instruction	After Instruction	
WREG	8823	8823	
RAM400	0823	0823	
SR	0000	0002	(Z=1)

Example 2

```
CP      0x1200    ; Compare (0x1200) with WREG (Word mode)
```

	Before Instruction	After Instruction	
WREG	2377	2377	
Data 1200	2277	2277	
SR	0000	0008	(N=1)

CP

Compare Wb with lit5, Set Status Flags

Syntax:	{label:} CP{.B} Wb, #lit5
Operands:	Wb ∈ [W0 ... W15] lit5 ∈ [0 ... 31]
Operation:	(Wb) – lit5
Status Affected:	DC, N, OV, Z, C
Encoding:	`1110` `0001` `0www` `wB00` `011k` `kkkk`
Description:	Compute (Wb) – lit5, and update the Status register. This instruction is equivalent to the `SUB` instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb. The ‘w’ bits select the address of the Wb base register. The ‘B’ bit selects byte or word operation (0 for word, 1 for byte). The ‘k’ bits provide the literal operand, a five-bit integer number. **Note:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
Words:	1
Cycles:	1

Example 1

```
CP.B  W4, #0x12        ; Compare W4 with 0x12 (Byte mode)
```

	Before Instruction	After Instruction	
W4	7711	7711	
SR	0000	0008	(N=1)

Example 2

```
CP    W4, #0x12        ; Compare W4 with 0x12 (Word mode)
```

	Before Instruction	After Instruction
W4	7713	7713
SR	0000	0000

CP

Compare Wb with Ws, Set Status Flags

Syntax:	{label:}	CP{.B}	Wb,	Ws
				[Ws]
				[Ws++]
				[Ws--]
				[++Ws]
				[--Ws]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]

Operation: (Wb) – (Ws)

Status Affected: DC, N, OV, Z, C

Encoding:

1110	0001	0www	wB00	0ppp	ssss

Description: Compute (Wb) – (Ws), and update the Status register. This instruction is equivalent to the `SUB` instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Ws.

The 'w' bits select the address of the Wb source register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'p' bits select the source Address mode.
The 's' bits select the address of the Ws source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
CP.B  W0, [W1++]     ; Compare [W1] with W0 (Byte mode)
                     ; Post-increment W1
```

	Before Instruction		After Instruction	
W0	ABA9	W0	ABA9	
W1	2000	W1	2001	
Data 2000	D004	Data 2000	D004	
SR	0000	SR	0008	(N=1)

Example 2

```
CP    W5, W6         ; Compare W6 with W5 (Word mode)
```

	Before Instruction		After Instruction	
W5	2334	W5	2334	
W6	8001	W6	8001	
SR	0000	SR	000C	(N,OV=1)

 Advance Information

CP0 — Compare f with 0x0, Set Status Flags

Syntax:	{label:} CP0{.B} f
Operands:	f ∈ [0 ... 8191]
Operation:	(f) – 0x0
Status Affected:	DC, N, OV, Z, C
Encoding:	`1110` `0010` `0B0f` `ffff` `ffff` `ffff`
Description:	Compute (f) – `0x0` and update the Status register. The result of the subtraction is not stored.

1110	0010	0B0f	ffff	ffff	ffff

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘f’ bits select the address of the file register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
CP0.B    RAM100    ; Compare RAM100 with 0x0 (Byte mode)
```

	Before Instruction	After Instruction	
RAM100	44C3	44C3	
SR	0000	0008	(N=1)

Example 2

```
CP0    0x1FFE    ; Compare (0x1FFE) with 0x0 (Word mode)
```

	Before Instruction	After Instruction
Data 1FFE	0001	0001
SR	0000	0000

CP0 — Compare Ws with 0x0, Set Status Flags

Syntax:	{label:}	CP0{.B}	Ws
			[Ws]
			[Ws++]
			[Ws--]
			[++Ws]
			[--Ws]

Operands: Ws ∈ [W0 ... W15]

Operation: (Ws) – 0x0000

Status Affected: DC, N, OV, Z, C

Encoding:

1110	0000	0000	0B00	0ppp	ssss

Description: Compute (Ws) – 0x0000 and update the Status register. The result of the subtraction is not stored. Register direct or indirect addressing may be used for Ws.

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'p' bits select the source Address mode.
The 's' bits select the address of the Ws source register.

> **Note:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
CP0.B  [W4--]      ; Compare [W4] with 0 (Byte mode)
                   ; Post-decrement W4
```

	Before Instruction	After Instruction	
W4	1001	1000	
Data 1000	0034	0034	
SR	0000	0002	(Z=1)

Example 2

```
CP0    [--W5]      ; Compare [--W5] with 0 (Word mode)
```

	Before Instruction	After Instruction	
W5	2400	23FE	
Data 23FE	9000	9000	
SR	0000	0008	(N=1)

CPB — Compare f with WREG using Borrow, Set Status Flags

Syntax:	{label:} CPB{.B} f
Operands:	f ∈ [0 ...8191]
Operation:	(f) – (WREG) – ($\overline{C}$)
Status Affected:	DC, N, OV, Z, C

Encoding:

1110	0011	1B0f	ffff	ffff	ffff

Description: Compute (f) – (WREG) – ($\overline{C}$), and update the Status register. This instruction is equivalent to the `SUBB` instruction, but the result of the subtraction is not stored.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘f’ bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The WREG is set to working register W0.
3: The Z flag is “sticky” for `ADDC, CPB, SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
CPB.B   RAM400  ; Compare RAM400 with WREG using C̅ (Byte mode)
```

	Before Instruction		After Instruction	
WREG	8823	WREG	8823	
RAM400	0823	RAM400	0823	
SR	0000	SR	0008	(N=1)

Example 2

```
CPB     0x1200  ; Compare (0x1200) with WREG using C̅ (Word mode)
```

	Before Instruction			After Instruction	
WREG	2377		WREG	2377	
Data 1200	2377		Data 1200	2377	
SR	0001	(C=1)	SR	0001	(C=1)

CPB — Compare Wb with lit5 using Borrow, Set Status Flags

Syntax:	{label:} CPB{.B} Wb, #lit5
Operands:	Wb ∈ [W0 ... W15] lit5 ∈ [0 ... 31]
Operation:	(Wb) – lit5 – ($\overline{C}$)
Status Affected:	DC, N, OV, Z, C
Encoding:	1110 0001 1www wB00 011k kkkk

1110	0001	1www	wB00	011k	kkkk

Description: Compute (Wb) – lit5 – ($\overline{C}$), and update the Status register. This instruction is equivalent to the `SUBB` instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb.

The ‘w’ bits select the address of the Wb source register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘k’ bits provide the literal operand, a five bit integer number.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The Z flag is “sticky” for `ADDC, CPB, SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
CPB.B  W4, #0x12    ; Compare W4 with 0x12 using C̄ (Byte mode)
```

	Before Instruction		After Instruction	
W4	7711		7711	
SR	0001	(C=1)	0008	(N=1)

Example 2

```
CPB.B  W4, #0x12    ; Compare W4 with 0x12 using C̄ (Byte mode)
```

	Before Instruction		After Instruction	
W4	7711		7711	
SR	0000		0008	(N=1)

Example 3

```
CPB    W12, #0x1F   ; Compare W12 with 0x1F using C̄ (Word mode)
```

	Before Instruction		After Instruction	
W12	0020		0020	
SR	0002	(Z=1)	0003	(Z, C=1)

Example 4

```
CPB    W12, #0x1F   ; Compare W12 with 0x1F using C̄ (Word mode)
```

	Before Instruction		After Instruction	
W12	0020		0020	
SR	0003	(Z, C=1)	0001	(C=1)

CPB

Compare Ws with Wb using Borrow, Set Status Flags

Syntax:	{label:}	CPB{.B}	Wb,	Ws
				[Ws]
				[Ws++]
				[Ws--]
				[++Ws]
				[--Ws]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]

Operation: (Wb) – (Ws) – ($\overline{C}$)

Status Affected: DC, N, OV, Z, C

Encoding:

1110	0001	1www	wB00	0ppp	ssss

Description: Compute (Wb) – (Ws) – ($\overline{C}$), and update the Status register. This instruction is equivalent to the `SUBB` instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Ws.

The 'w' bits select the address of the Wb source register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'p' bits select the source Address mode.
The 's' bits select the address of the Ws source register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The Z flag is "sticky" for `ADDC`, `CPB`, `SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
CPB.B  W0, [W1++] ; Compare [W1] with W0 using C̅ (Byte mode)
                  ; Post-increment W1
```

	Before Instruction		After Instruction	
W0	ABA9		ABA9	
W1	1000		1001	
Data 1000	D0A9		D0A9	
SR	0002	(Z=1)	0008	(N=1)

Example 2

```
CPB.B  W0, [W1++] ; Compare [W1] with W0 using C̅ (Byte mode)
                  ; Post-increment W1
```

	Before Instruction		After Instruction	
W0	ABA9		ABA9	
W1	1000		1001	
Data 1000	D0A9		D0A9	
SR	0001	(C=1)	0001	(C=1)

Example 3 `CPB W4, W5 ; Compare W5 with W4 using` $\overline{C}$ `(Word mode)`

	Before Instruction			After Instruction	
W4	4000		W4	4000	
W5	3000		W5	3000	
SR	0001	(C=1)	SR	0001	(C=1)

CPSEQ

Compare Wb with Wn, Skip if Equal (Wb = Wn)

Syntax:	{label:}	CPSEQ{.B}	Wb,	Wn

Operands: Wb ∈ [W0 ... W15]
Wn ∈ [W0 ... W15]

Operation: (Wb) – (Wn)
Skip if (Wb) = (Wn)

Status Affected: None

Encoding:

1110	0111	1www	wB00	0000	ssss

Description: Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) = (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a `NOP` is executed instead. If (Wb) ≠ (Wn), the next instruction is executed as normal.

The ‘w’ bits select the address of the Wb source register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘s’ bits select the address of the Ws source register.

> **Note:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1 (2 or 3 if skip taken)

Example 1

```
002000 HERE:  CPSEQ.B W0, W1   ; If W0 = W1 (Byte mode),
002002        GOTO    BYPASS   ; skip the GOTO
002004        . . .
002006        . . .
002008 BYPASS:. . .
00200A        . . .
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 2002
W0	1001	W0	1001
W1	1000	W1	1000
SR	0000	SR	0000

Example 2

```
018000 HERE:   CPSEQ  W4, W8   ; If W4 = W8 (Word mode),
018002         CALL   _FIR     ; skip the subroutine call
018006         ...
018008         ...
```

	Before Instruction		After Instruction
PC	01 8000	PC	01 8006
W4	3344	W4	3344
W8	3344	W8	3344
SR	0002 (Z=1)	SR	0002 (Z=1)

CPSGT

Signed Compare Wb with Wn, Skip if Greater Than (Wb > Wn)

Syntax:	{label:} CPSGT{.B} Wb, Wn
Operands:	Wb ∈ [W0 ... W15] Wn ∈ [W0 ... W15]
Operation:	(Wb) – (Wn) Skip if (Wb) > (Wn)
Status Affected:	None

Encoding:

1110	0110	0www	wB00	0000	ssss

Description: Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) > (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a `NOP` is executed instead. Otherwise, the next instruction is executed as normal.

The 'w' bits select the address of the Wb source register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 's' bits select the address of the Ws source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1 (2 or 3 if skip taken)

Example 1

```
002000 HERE:  CPSGT.B   W0, W1 ; If W0 > W1 (Byte mode),
002002        GOTO      BYPASS ; skip the GOTO
002006        . . .
002008        . . .
00200A BYPASS . . .
00200C        . . .
```

	Before Instruction		After Instruction	
PC	00 2000		00 2006	
W0	00FF		00FF	
W1	26FE		26FE	
SR	0009	(N, C=1)	0009	(N, C=1)

Example 2

```
018000 HERE:  CPSGT     W4, W5 ; If W4 > W5 (Word mode),
018002        CALL      _FIR   ; skip the subroutine call
018006        ...
018008        ...
```

	Before Instruction		After Instruction	
PC	01 8000		01 8002	
W4	2600		2600	
W5	2600		2600	
SR	0004	(OV=1)	0004	(OV=1)

CPSLT

Signed Compare Wb with Wn, Skip if Less Than (Wb < Wn)

Syntax:	{label:} CPSLT{.B} Wb, Wn
Operands:	Wb ∈ [W0 ... W15] Wn ∈ [W0 ... W15]
Operation:	(Wb) – (Wn) Skip if (Wb) < (Wn)
Status Affected:	None
Encoding:	1110 0110 1www wB00 0000 ssss

Encoding:					
1110	0110	1www	wB00	0000	ssss

Description: Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) < (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a `NOP` is executed instead. Otherwise, the next instruction is executed as normal.

The ‘w’ bits select the address of the Wb source register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘s’ bits select the address of the Ws source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1 (2 or 3 if skip taken)

Example 1

```
002000 HERE:   CPSLT.B   W8, W9 ; If W8 < W9 (Byte mode),
002002         GOTO      BYPASS ; skip the GOTO
002006         . . .
002008         . . .
00200A BYPASS: . . .
00200C         . . .
```

	Before Instruction		After Instruction	
PC	00 2000		00 2002	
W8	00FF		00FF	
W9	26FE		26FE	
SR	0008	(N=1)	0008	(N=1)

Example 2

```
018000 HERE:  CPSLT    W3, W6  ; If W3 < W6 (Word mode),
018002        CALL     _FIR    ; skip the subroutine call
018006          . . .
018008          . . .
```

	Before Instruction	After Instruction
PC	01 8000	01 8006
W3	2600	2600
W6	3000	3000
SR	0000	0000

CPSNE

Signed Compare Wb with Wn, Skip if Not Equal (Wb ≠ Wn)

Syntax:	{label:} CPSNE{.B} Wb, Wn
Operands:	Wb ∈ [W0 ... W15] Wn ∈ [W0 ... W15]
Operation:	(Wb) – (Wn) Skip if (Wb) ≠ (Wn)
Status Affected:	None

Encoding:

1110	0111	0www	wB00	0000	ssss

Description: Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) ≠ (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a `NOP` is executed instead. Otherwise, the next instruction is executed as normal.

The 'w' bits select the address of the Wb source register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 's' bits select the address of the Ws source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1 (2 or 3 if skip taken)

Example 1

```
002000 HERE:   CPSNE.B  W2, W3 ; If W2 != W3 (Byte mode),
002002         GOTO     BYPASS ; skip the GOTO
002006         . . .
002008         . . .
00200A BYPASS: . . .
00200C         . . .
```

	Before Instruction		After Instruction
PC	00 2000	PC	00 2006
W2	00FF	W2	00FF
W3	26FE	W3	26FE
SR	0001 (C=1)	SR	0001 (C=1)

Example 2

```
018000 HERE:   CPSNE  W0, W8   ; If W0 != W8 (Word mode),
018002         CALL   _FIR     ; skip the subroutine call
018006         ...
018008         ...
```

	Before Instruction		After Instruction
PC	01 8000	PC	01 8002
W0	3000	W0	3000
W8	3000	W8	3000
SR	0000	SR	0000

DAW.B — Decimal Adjust Wn

Syntax: {label:} DAW.B Wn

Operands: Wn ∈ [W0 ... W15]

Operation:

```
If (Wn<3:0> > 9) or (DC = 1)
   (Wn<3:0>) + 6 → Wn<3:0>
Else
   (Wn<3:0>) → Wn<3:0>

If (Wn<7:4> > 9) or (C = 1)
   (Wn<7:4>) + 6 → Wn<7:4>
Else
   (Wn<7:4>) → Wn<7:4>
```

Status Affected: C

Encoding:

1111	1101	0100	0000	0000	ssss

Description: Adjust the Least Significant Byte in Wn to produce a binary coded decimal (BCD) result. The Most Significant Byte of Wn is not changed, and the Carry flag is used to indicate any decimal rollover. Register direct addressing must be used for Wn.

The 's' bits select the address of the source/destination register.

Note 1: This instruction is used to correct the data format after two packed BCD bytes have been added.

2: This instruction operates in Byte mode only and the `.B` extension must be included with the opcode.

Words: 1

Cycles: 1

Example 1

```
DAW.B  W0       ; Decimal adjust W0
```

	Before Instruction	After Instruction
W0	771A	7720
SR	0002 (DC=1)	0002 (DC=1)

Example 2

```
DAW.B  W3       ; Decimal adjust W3
```

	Before Instruction	After Instruction
W3	77AA	7710
SR	0000	0001 (C=1)

DEC — Decrement f

Syntax:	{label:}	DEC{.B}	f	{,WREG}	

Operands: f ∈ [0 ... 8191]

Operation: (f) – 1 → destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1101	0BDf	ffff	ffff	ffff

Description: Subtract one from the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘D’ bit selects the destination (`0` for WREG, `1` for file register).
The ‘f’ bits select the address of the file register.

> **Note 1:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
>
> **2:** The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
DEC.B  0x200          ; Decrement (0x200) (Byte mode)
```

	Before Instruction		After Instruction	
Data 200	80FF	Data 200	80FE	
SR	0000	SR	0009	(N,C=1)

Example 2

```
DEC    RAM400, WREG   ; Decrement RAM400 and store to WREG
                      ; (Word mode)
```

	Before Instruction		After Instruction
WREG	1211	WREG	0822
RAM400	0823	RAM400	0823
SR	0000	SR	0000

 Advance Information

DEC

Decrement Ws

Syntax:	{label:}	DEC{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Ws) – 1 → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1001	0Bqq	qddd	dppp	ssss

Description: Subtract one from the contents of the source register Ws and place the result in the destination register Wd. Either register direct or indirect addressing may be used by Ws and Wd.

The ‘B’ bit selects byte or word operation (0 for word, 1 for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
DEC.B  [W7++], [W8++]   ; DEC [W7] and store to [W8] (Byte mode)
                         ; Post-increment W7, W8
```

	Before Instruction		After Instruction
W7	2301	W7	2302
W8	2400	W8	2401
Data 2300	5607	Data 2300	5607
Data 2400	ABCD	Data 2400	AB55
SR	0000	SR	0000

Example 2

```
DEC  W5, [W6++]   ; Decrement W5 and store to [W6] (Word mode)
                  ; Post-increment W6
```

	Before Instruction		After Instruction
W5	D004	W5	D004
W6	2000	W6	2002
Data 2000	ABA9	Data 2000	D003
SR	0000	SR	0009 (N, C=1)

DEC2 — Decrement f by 2

Syntax:	{label:}	DEC2{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (f) – 2 → destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1101	1BDf	ffff	ffff	ffff

Description: Subtract two from the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'D' bit selects the destination (0 for WREG, 1 for file register).
The 'f' bits select the address of the file register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
DEC2.B   0x200         ; Decrement (0x200) by 2 (Byte mode)
```

	Before Instruction	After Instruction	
Data 200	80FF	80FD	
SR	0000	0009	(N, C=1)

Example 2

```
DEC2     RAM400, WREG  ; Decrement RAM400 by 2 and
                       ; store to WREG (Word mode)
```

	Before Instruction	After Instruction
WREG	1211	0821
RAM400	0823	0823
SR	0000	0000

 Advance Information

DEC2 — Decrement Ws by 2

Syntax:	{label:}	DEC2{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Ws) – 2 → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1001	1Bqq	qddd	dppp	ssss

Description: Subtract two from the contents of the source register Ws and place the result in the destination register Wd. Either register direct or indirect addressing may be used by Ws and Wd.

The ‘B’ bit selects byte or word operation (0 for word, 1 for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
DEC2.B [W7--], [W8--]; DEC [W7] by 2, store to [W8] (Byte mode)
                      ; Post-decrement W7, W8
```

	Before Instruction	After Instruction
W7	2301	2300
W8	2400	23FF
Data 2300	0107	0107
Data 2400	ABCD	ABFF
SR	0000	0008 (N=1)

Example 2

```
DEC2  W5, [W6++]  ; DEC W5 by 2, store to [W6] (Word mode)
                  ; Post-increment W6
```

	Before Instruction	After Instruction
W5	D004	D004
W6	1000	1002
Data 1000	ABA9	D002
SR	0000	0009 (N, C=1)

DISI

Disable Interrupts Temporarily

Syntax:	{label:} DISI #lit14
Operands:	lit14 ∈ [0 ... 16383]
Operation:	lit14 → DISICNT 1 → DISI Disable interrupts for (lit14+1) cycles
Status Affected:	None
Encoding:	1111 1100 00kk kkkk kkkk kkkk
Description:	Disable interrupts of priority 0 through priority 6 for (lit14+1) instruction cycles after the instruction executes. The lit14 value is written to the DISICNT register, and the DISI flag (INTCON2<14>) is set to ‘1’. This instruction can be used before executing time critical code, to limit the effects of interrupts.

> **Note:** This instruction does not prevent priority 7 interrupts and traps from running. See the dsPIC30F Data Sheet for details.

Words:	1
Cycles:	1

Example 1

```
002000 HERE:   DISI  #100  ; Disable interrupts for 101 cycles
002002                     ; now execute time critical code
002004         . . .
```

	Before Instruction	After Instruction	
PC	00 2000	00 2002	
DISICNT	0000	0100	
INTCON2	0000	4000	(DISI=1)
SR	0000	0000	

 Advance Information

DIV.S

Signed Integer Divide

Syntax:	{label:}	DIV.S{W}	Wm, Wn
		DIV.SD	Wm, Wn

Operands: Wm ∈ [W0 ... W15] for word operation
Wm ∈ [W0, W2, W4 ... W14] for double operation
Wn ∈ [W2 ... W15]

Operation: For word operation (default):
Wm → W0
If (Wm<15> = 1):
0xFFFF → W1
Else:
0x0 → W1
W1:W0 / Wn → W0
Remainder → W1

For double operation (DIV.SD):
Wm+1:Wm → W1:W0
W1:W0 / Wn → W0
Remainder → W1

Status Affected: N, OV, Z, C

Encoding:

1101	1000	0ttt	tvvv	vW00	ssss

Description: Iterative, signed integer divide, where the dividend is stored in Wm (for a 16-bit by 16-bit divide) or Wm+1:Wm (for a 32-bit by 16-bit divide) and the divisor is stored in Wn. In the default word operation, Wm is first copied to W0 and sign-extended through W1 to perform the operation. In the double operation, Wm+1:Wm is first copied to W1:W0. The 16-bit quotient of the divide operation is stored in W0, and the 16-bit remainder is stored in W1.

This instruction must be executed 18 times using the `REPEAT` instruction (with an iteration count of 17) to generate the correct quotient and remainder. The N flag will be set if the remainder is negative and cleared otherwise. The OV flag will be set if the divide operation resulted in an overflow and cleared otherwise. The Z flag will be set if the remainder is 0 and cleared otherwise. The C flag is used to implement the divide algorithm and its final value should not be used.

The ‘t’ bits select the Most Significant Word of the dividend for the double operation. These bits are clear for the word operation.
The ‘v’ bits select the Least Significant Word of the dividend.
The ‘W’ bit selects the dividend size (0 for 16-bit, 1 for 32-bit).
The ‘s’ bits select the divisor register.

Note 1: The extension `.D` in the instruction denotes a double-word (32-bit) dividend rather than a word dividend. You may use a `.W` extension to denote a word operation, but it is not required.

2: Unexpected results will occur if the quotient can not be represented in 16 bits. When this occurs for the double operation (`DIV.SD`), the OV status bit will be set and the quotient and remainder should not be used. For the word operation (`DIV.S`), only one type of overflow may occur (0x8000 / 0xFFFF = +32768 or 0x00008000), which allows the OV status bit to interpret the result.

3: Dividing by zero will initiate an arithmetic error trap during the first cycle of execution.

4: This instruction is interruptible on each instruction cycle boundary.

DIV.S

Signed Integer Divide

Words: 1

Cycles: 18 (plus 1 for `REPEAT` execution)

Example 1

```
REPEAT #17        ; Execute DIV.S 18 times
DIV.S  W3, W4     ; Divide W3 by W4
                  ; Store quotient to W0, remainder to W1
```

	Before Instruction	After Instruction
W0	5555	013B
W1	1234	0003
W3	3000	3000
W4	0027	0027
SR	0000	0000

Example 2

```
REPEAT  #17       ; Execute DIV.SD 18 times
DIV.SD  W0, W12   ; Divide W1:W0 by W12
                  ; Store quotient to W0, remainder to W1
```

	Before Instruction	After Instruction
W0	2500	FA6B
W1	FF42	EF00
W12	2200	2200
SR	0000	0008 (N=1)

DIV.U

Unsigned Integer Divide

Syntax:	{label:}	DIV.U{W}	Wm, Wn
		DIV.UD	Wm, Wn

Operands: Wm ∈ [W0 ... W15] for word operation
Wm ∈ [W0, W2, W4 ... W14] for double operation
Wn ∈ [W2 ... W15]

Operation: For word operation (default):
Wm → W0
0x0 → W1
W1:W0 / Wn → W0
Remainder → W1

For double operation (DIV.UD):
Wm+1:Wm → W1:W0
W1:W0 / Wns → W0
Remainder → W1

Status Affected: N, OV, Z, C

Encoding:

1101	1000	1ttt	tvvv	vW00	ssss

Description: Iterative, unsigned integer divide, where the dividend is stored in Wm (for a 16-bit by 16-bit divide), or Wm+1:Wm (for a 32-bit by 16-bit divide) and the divisor is stored in Wn. In the word operation, Wm is first copied to W0 and W1 is cleared to perform the divide. In the double operation, Wm+1:Wm is first copied to W1:W0. The 16-bit quotient of the divide operation is stored in W0, and the 16-bit remainder is stored in W1.

This instruction must be executed 18 times using the `REPEAT` instruction (with an iteration count of 17) to generate the correct quotient and remainder. The N flag will always be cleared. The OV flag will be set if the divide operation resulted in an overflow and cleared otherwise. The Z flag will be set if the remainder is 0 and cleared otherwise. The C flag is used to implement the divide algorithm and its final value should not be used.

The ‘t’ bits select the Most Significant Word of the dividend for the double operation. These bits are clear for the word operation.
The ‘v’ bits select the Least Significant Word of the dividend.
The ‘W’ bit selects the dividend size (0 for 16-bit, 1 for 32-bit).
The ‘s’ bits select the divisor register.

Note 1: The extension `.D` in the instruction denotes a double-word (32-bit) dividend rather than a word dividend. You may use a `.W` extension to denote a word operation, but it is not required.
2: Unexpected results will occur if the quotient can not be represented in 16 bits. This may only occur for the double operation (`DIV.UD`). When an overflow occurs, the OV status bit will be set and the quotient and remainder should not be used.
3: Dividing by zero will initiate an arithmetic error trap during the first cycle of execution.
4: This instruction is interruptible on each instruction cycle boundary.

Words: 1

Cycles: 18 (plus 1 for `REPEAT` execution)

Example 1

```
REPEAT #17      ; Execute DIV.U 18 times
DIV.U  W2, W4   ; Divide W2 by W4
                ; Store quotient to W0, remainder to W1
```

	Before Instruction	After Instruction	
W0	5555	0040	
W1	1234	0000	
W2	8000	8000	
W4	0200	0200	
SR	0000	0002	(Z=1)

Example 2

```
REPEAT  #17        ; Execute DIV.UD 18 times
DIV.UD  W10, W12   ; Divide W11:W10 by W12
                   ; Store quotient to W0, remainder to W1
```

	Before Instruction	After Instruction
W0	5555	01F2
W1	1234	0100
W10	2500	2500
W11	0042	0042
W12	2200	2200
SR	0000	0000

 Advance Information

DIVF

Fractional Divide

Syntax:	{label:}	DIVF	Wm, Wn

Operands: Wm ∈ [W0 ... W15]
Wn ∈ [W2 ... W15]

Operation: 0x0 → W0
Wm → W1
W1:W0 / Wn → W0
Remainder → W1

Status Affected: N, OV, Z, C

Encoding:

1101	1001	0ttt	t000	0000	ssss

Description: Iterative, signed fractional 16-bit by 16-bit divide, where the dividend is stored in Wm and the divisor is stored in Wn. To perform the operation, W0 is first cleared and Wm is copied to W1. The 16-bit quotient of the divide operation is stored in W0, and the 16-bit remainder is stored in W1. The sign of the remainder will be the same as the sign of the dividend.

This instruction must be executed 18 times using the `REPEAT` instruction (with an iteration count of 17) to generate the correct quotient and remainder. The N flag will be set if the remainder is negative and cleared otherwise. The OV flag will be set if the divide operation resulted in an overflow and cleared otherwise. The Z flag will be set if the remainder is 0 and cleared otherwise. The C flag is used to implement the divide algorithm and its final value should not be used.

The 't' bits select the dividend register.
The 's' bits select the divisor register.

Note 1: For the fractional divide to be effective, Wm must be less than or equal to Wn. If Wm is greater than Wn, unexpected results will occur because the fractional result will be greater than 1.0. When this occurs, the OV status bit will be set and the quotient and remainder should not be used.

2: Dividing by zero will initiate an arithmetic error trap during the first cycle of execution.

3: This instruction is interruptible on each instruction cycle boundary.

Words: 1

Cycles: 18 (plus 1 for `REPEAT` execution)

Example 1

```
REPEAT  #17      ; Execute DIVF 18 times
DIVF    W8, W9   ; Divide W8 by W9
                 ; Store quotient to W0, remainder to W1
```

	Before Instruction		After Instruction	
W0	8000	W0	2000	
W1	1234	W1	0000	
W8	1000	W8	1000	
W9	4000	W9	4000	
SR	0000	SR	0002	(Z=1)

Example 2

```
REPEAT #17       ; Execute DIVF 18 times
DIVF   W8, W9    ; Divide W8 by W9
                 ; Store quotient to W0, remainder to W1
```

	Before Instruction	After Instruction	
W0	8000	F000	
W1	1234	0000	
W8	1000	1000	
W9	8000	8000	
SR	0000	0002	(Z=1)

Example 3

```
REPEAT #17       ; Execute DIVF 18 times
DIVF   W0, W1    ; Divide W0 by W1
                 ; Store quotient to W0, remainder to W1
```

	Before Instruction	After Instruction	
W0	8002	7FFE	
W1	8001	8002	
SR	0000	0008	(N=1)

DO

Initialize Hardware Loop Literal

Syntax:	{label:}	DO	#lit14,	Expr

Operands: lit14 ∈ [0 ... 16383]
Expr may be an absolute address, label or expression.
Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation: Push DO shadows (DCOUNT, DOEND, DOSTART)
(lit14) → DCOUNT
(PC)+4 → PC
(PC) → DOSTART
(PC) + (2*Slit16) → DOEND
Increment DL<2:0> (CORCON<10:8>)

Status Affected: DA

Encoding:

0000	1000	00kk	kkkk	kkkk	kkkk
0000	0000	nnnn	nnnn	nnnn	nnnn

Description: Initiate a no overhead hardware DO loop, which is executed (lit14+1) times. The DO loop begins at the address following the `DO` instruction, and ends at the address 2*Slit16 instruction words away. The 14-bit count value (lit14) supports a maximum loop count value of 16384, and the 16-bit offset value (Slit16) supports offsets of 32K instruction words in both directions.

When this instruction executes, DCOUNT, DOSTART and DOEND are first pushed into their respective shadow registers, and then updated with the new DO loop parameters specified by the instruction. The DO level count, DL<2:0> (CORCON<8:10>), is then incremented. After the DO loop completes execution, the pushed DCOUNT, DOSTART and DOEND registers are restored, and DL<2:0> is decremented.

The ‘k’ bits specify the loop count.
The ‘n’ bits are a signed literal that specifies the number of instructions offset from the PC to the last instruction executed in the loop.

Special Features, Restrictions:
The following features and restrictions apply to the `DO` instruction.

1. Using a loop count of 0 will result in the loop being executed one time.
2. Using a loop size of -2, -1 or 0 is invalid. Unexpected results may occur if these offsets are used.
3. The very **last two** instructions of the DO loop can NOT be:
   - an instruction which changes program control flow
   - a `DO` or `REPEAT` instruction

   Unexpected results may occur if any of these instructions are used.

Note 1: The `DO` instruction is interruptible and supports 1 level of hardware nesting. Nesting up to an additional 5 levels may be provided in software by the user. See the dsPIC30F Family Reference Manual for details.

2: The linker will convert the specified expression into the offset to be used.

Words: 2

Cycles: 2

Example 1

```
002000 LOOP6:   DO    #5, END6     ; Initiate DO loop (5 reps)
002004          ADD   W1, W2, W3   ; First instruction in loop
002006          . . .
002008          . . .
00200A END6:    SUB   W2, W3, W4   ; Last instruction in loop
00200C          . . .
```

	Before Instruction			After Instruction	
PC	00 2000		PC	00 2004	
DCOUNT	0000		DCOUNT	0005	
DOSTART	FF FFFF		DOSTART	00 2004	
DOEND	FF FFFF		DOEND	00 200A	
CORCON	0000		CORCON	0100	(DL=1)
SR	0001	(C=1)	SR	0201	(DA, C=1)

Example 2

```
01C000 LOOP12: DO  #0x160, END12  ; Init DO loop (352 reps)
01C004         DEC  W1, W2        ; First instruction in loop
01C006          . . .
01C008          . . .
01C00A          . . .
01C00C          . . .
01C00E          . . .
01C010         CALL  _FIR88       ; Call the FIR88 subroutine
01C014 END12:  NOP                ; Last instruction in loop
                                  ; (Required NOP filler)
```

	Before Instruction			After Instruction	
PC	01 C000		PC	01 C004	
DCOUNT	0000		DCOUNT	0160	
DOSTART	FF FFFF		DOSTART	01 C004	
DOEND	FF FFFF		DOEND	01 C014	
CORCON	0000		CORCON	0100	(DL=1)
SR	0008	(N=1)	SR	0208	(DA, N=1)

 Advance Information

DO

Initialize Hardware Loop Wn

Syntax:	{label:}	DO	Wn,	Expr

Operands: Wn ∈ [W0 ... W15]
Expr may be an absolute address, label or expression.
Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation: Push Shadows (DCOUNT, DOEND, DOSTART)
(Wn) → DCOUNT
(PC)+4 → PC
(PC) → DOSTART
(PC) + (2*Slit16) → DOEND
Increment DL<2:0> (CORCON<10:8>)

Status Affected: DA

Encoding:

0000	1000	1000	0000	0000	ssss
0000	0000	nnnn	nnnn	nnnn	nnnn

Description: Initiate a no overhead hardware DO loop, which is executed (Wn+1) times. The DO loop begins at the address following the `DO` instruction, and ends at the address 2*Slit16 instruction words away. The lower 14 bits of Wn support a maximum count value of 16384, and the 16-bit offset value (Slit16) supports offsets of 32K instruction words in both directions.

When this instruction executes, DCOUNT, DOSTART and DOEND are first pushed into their respective shadow registers, and then updated with the new DO loop parameters specified by the instruction. The DO level count, DL<2:0> (CORCON<8:10>), is then incremented. After the DO loop completes execution, the pushed DCOUNT, DOSTART and DOEND registers are restored, and DL<2:0> is decremented.

The ‘s’ bits specify the register Wn that contains the loop count.
The ‘n’ bits are a signed literal that specifies the number of instructions offset from (PC+4), which is the last instruction executed in the loop.

Special Features, Restrictions:
The following features and restrictions apply to the `DO` instruction.

1. Using a loop count of 0 will result in the loop being executed one time.
2. Using an offset of -2, -1 or 0 is invalid. Unexpected results may occur if these offsets are used.
3. The very **last two** instructions of the DO loop can NOT be:
   - an instruction which changes program control flow
   - a `DO` or `REPEAT` instruction

   Unexpected results may occur if these last instructions are used.

Note 1: The `DO` instruction is interruptible and supports 1 level of nesting. Nesting up to an additional 5 levels may be provided in software by the user. See the dsPIC30F Family Reference Manual for details.

2: The linker will convert the specified expression into the offset to be used.

Words: 2

Cycles: 2

Example 1

```
002000 LOOP6:   DO      W0, END6   ; Initiate DO loop (W0 reps)
002004          ADD     W1, W2, W3 ; First instruction in loop
002006          . . .
002008          . . .
00200A          . . .
00200C          REPEAT #6
00200E          SUB     W2, W3, W4
002010 END6:    NOP                ; Last instruction in loop
                                   ; (Required NOP filler)
```

	Before Instruction		After Instruction	
PC	00 2000	PC	00 2004	
W0	0012	W0	0012	
DCOUNT	0000	DCOUNT	0012	
DOSTART	FF FFFF	DOSTART	00 2004	
DOEND	FF FFFF	DOEND	00 2010	
CORCON	0000	CORCON	0100	(DL=1)
SR	0000	SR	0080	(DA=1)

Example 2

```
002000 LOOPA:   DO      W7, ENDA   ; Initiate DO loop (W7 reps)
002004          SWAP    W0         ; First instruction in loop
002006          . . .
002008          . . .
00200A          . . .
002010 ENDA:    MOV     W1, [W2++] ; Last instruction in loop
```

	Before Instruction		After Instruction	
PC	00 2000	PC	00 2004	
W7	E00F	W7	E00F	
DCOUNT	0000	DCOUNT	200F	
DOSTART	FF FFFF	DOSTART	00 2004	
DOEND	FF FFFF	DOEND	00 2010	
CORCON	0000	CORCON	0100	(DL=1)
SR	0000	SR	0080	(DA=1)

ED — Euclidean Distance (No Accumulate)

Syntax:	{label:} ED	Wm*Wm,	Acc,	[Wx],	[Wy],	Wxd
				[Wx]+=kx,	[Wy]+=ky,	
				[Wx]-=kx,	[Wy]-=ky,	
				[W9+W12],	[W11+W12],	

Operands:
Acc ∈ [A,B]
Wm*Wm ∈ [W4*W4, W5*W5, W6*W6, W7*W7]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]
Wxd ∈ [W4 ... W7]

Operation:
(Wm)*(Wm) → Acc(A or B)
([Wx]–[Wy])→ Wxd
(Wx)+kx→Wx
(Wy)+ky→Wy

Status Affected: OA, OB, OAB

Encoding:

1111	00mm	A1xx	00ii	iijj	jj11

Description: Compute the square of Wm, and optionally compute the difference of the pre-fetch values specified by [Wx] and [Wy]. The results of Wm*Wm are sign-extended to 40-bits and stored in the specified accumulator. The results of [Wx] – [Wy] are stored in Wxd, which may be the same as Wm.

Operands Wx, Wxd and Wyd specify the pre-fetch operations which support indirect and register offset addressing as described in **Section 4.14.1 "MAC Pre-Fetches"**.

The 'm' bits select the operand register Wm for the square.
The 'A' bit selects the accumulator for the result.
The 'x' bits select the pre-fetch difference Wxd destination.
The 'i' bits select the Wx pre-fetch operation.
The 'j' bits select the Wy pre-fetch operation.

Words: 1

Cycles: 1

Example 1

```
ED  W4*W4, A, [W8]+=2, [W10]-=2, W4; Square W4 to ACCA
                                   ; [W8]-[W10] to W4
                                   ; Post-increment W8
                                   ; Post-decrement W10
```

	Before Instruction		After Instruction
W4	009A	W4	0057
W8	1100	W8	1102
W10	2300	W10	22FE
ACCA	00 3D0A 0000	ACCA	00 0000 5CA4
Data 1100	007F	Data 1100	007F
Data 2300	0028	Data 2300	0028
SR	0000	SR	0000

Example 2

```
ED  W5*W5, B, [W9]+=2, [W11+W12], W5   ; Square W5 to ACCB
                                       ; [W11+W12]-[W9] to W5
                                       ; Post-increment W9
```

	Before Instruction		After Instruction
W5	43C2	W5	3F3F
W9	1200	W9	1202
W11	2500	W11	2500
W12	0008	W12	0008
ACCB	00 28E3 F14C	ACCB	00 11EF 1F04
Data 1200	6A7C	Data 1200	6A7C
Data 2508	2B3D	Data 2508	2B3D
SR	0000	SR	0000

EDAC

Euclidean Distance

Syntax:	{label:} EDAC	Wm*Wm,	Acc,	[Wx],	[Wy],	Wxd
				[Wx]+=kx,	[Wy]+=ky,	
				[Wx]-=kx,	[Wy]-=ky,	
				[W9+W12],	[W11+W12],	

Operands:
Acc ∈ [A,B]
Wm*Wm ∈ [W4*W4, W5*W5, W6*W6, W7*W7]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]
Wxd ∈ [W4 ... W7]

Operation:
(Acc(A or B)) + (Wm)*(Wm) → Acc(A or B)
([Wx]–[Wy])→ Wxd
(Wx)+kx→Wx
(Wy)+ky→Wy

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1111	00mm	A1xx	00ii	iijj	jj10

Description: Compute the square of Wm, and also the difference of the pre-fetch values specified by [Wx] and [Wy]. The results of Wm*Wm are sign-extended to 40-bits and added to the specified accumulator. The results of [Wx] – [Wy] are stored in Wxd, which may be the same as Wm.

Operands Wx, Wxd and Wyd specify the pre-fetch operations which support indirect and register offset addressing as described in **Section 4.14.1 “MAC Pre-Fetches”**.

The ‘m’ bits select the operand register Wm for the square.
The ‘A’ bit selects the accumulator for the result.
The ‘x’ bits select the pre-fetch difference Wxd destination.
The ‘i’ bits select the Wx pre-fetch operation.
The ‘j’ bits select the Wy pre-fetch operation.

Words: 1

Cycles: 1

Example 1

```
EDAC  W4*W4, A, [W8]+=2, [w10]-=2, W4    ; Square W4 and
                                         ; add to ACCA
                                         ; [W10]-[W8] to W4
                                         ; Post-increment W8
                                         ; Post-decrement W10
```

	Before Instruction		After Instruction
W4	009A	W4	0057
W8	1100	W8	1102
W10	2300	W10	22FE
ACCA	00 3D0A 3D0A	ACCA	00 3D0A 99AE
Data 1100	007F	Data 1100	007F
Data 2300	0028	Data 2300	0028
SR	0000	SR	0000

Example 2

```
EDAC W5*W5, B, [w9]+=2, [W11+W12], W5  ; Square W5 and
                                       ; add to ACCB
                                       ; [W9]-[W11+W12] to W5
                                       ; Post-increment W9
```

	Before Instruction		After Instruction
W5	43C2	W5	3F3F
W9	1200	W9	1202
W11	2500	W11	2500
W12	0008	W12	0008
ACCB	00 28E3 F14C	ACCB	00 3AD3 1050
Data 1200	6A7C	Data 1200	6A7C
Data 2508	2B3D	Data 2508	2B3D
SR	0000	SR	0000

 Advance Information

EXCH

Exchange Wns and Wnd

Syntax:	{label:}	EXCH	Wns,	Wnd

Operands: Wns ∈ [W0 ... W15]
Wnd ∈ [W0 ... W15]

Operation: (Wns) ↔ (Wnd)

Status Affected: None

Encoding:

1111	1101	0000	0ddd	d000	ssss

Description: Exchange the word contents of two working registers. Register direct addressing must be used for Wns and Wnd.

The 'd' bits select the address of the first register.
The 's' bits select the address of the second register.

Note: This instruction only executes in Word mode.

Words: 1

Cycles: 1

Example 1

```
EXCH  W1, W9      ; Exchange the contents of W1 and W9
```

	Before Instruction		After Instruction
W1	55FF	W1	A3A3
W9	A3A3	W9	55FF
SR	0000	SR	0000

Example 2

```
EXCH  W4, W5      ; Exchange the contents of W4 and W5
```

	Before Instruction		After Instruction
W4	ABCD	W4	4321
W5	4321	W5	ABCD
SR	0000	SR	0000

FBCL

Find First Bit Change from Left

Syntax:	{label:}	FBCL	Ws,	Wnd
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
Wnd ∈ [W0 ... W15]

Operation:

```
Max_Shift = 15
Sign = (Ws) & 0x8000
Temp = (Ws) << 1
Shift = 0
While ( (Shift < Max_Shift) && ( (Temp & 0x8000) == Sign) )
   Temp = Temp << 1
   Shift = Shift + 1
-Shift → (Wnd)
```

Status Affected: C

Encoding:

1101	1111	0000	0ddd	dppp	ssss

Description: Find the first occurrence of a one (for a positive value), or zero (for a negative value), starting from the Most Significant bit after the sign bit of Ws and working towards the Least Significant bit of the word operand. The bit number result is sign-extended to 16-bits and placed in Wnd.

The next Most Significant bit after the sign bit is allocated bit number 0 and the Least Significant bit is allocated bit number -14. This bit ordering allows for the immediate use of Wd with the `SFTAC` instruction for scaling values up. If a bit change is not found, a result of -15 is returned and the C flag is set. When a bit change is found, the C flag is cleared.

The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: 1

Example 1

```
FBCL  W1, W9        ; Find 1st bit change from left in W1
                    ; and store result to W9
```

	Before Instruction		After Instruction
W1	55FF	W1	55FF
W9	FFFF	W9	0000
SR	0000	SR	0000

 Advance Information

Example 2

```
FBCL  W1, W9        ; Find 1st bit change from left in W1
                    ; and store result to W9
```

	Before Instruction		After Instruction
W1	FFFF	W1	FFFF
W9	BBBB	W9	FFF1
SR	0000	SR	0001 (C=1)

Example 3

```
FBCL  [W1++], W9    ; Find 1st bit change from left in [W1]
                    ; and store result to W9
                    ; Post-increment W1
```

	Before Instruction		After Instruction
W1	2000	W1	2002
W9	BBBB	W9	FFF9
Data 2000	FF0A	Data 2000	FF0A
SR	0000	SR	0000

FF1L Find First One from Left

Syntax:	{label:}	FF1L	Ws,	Wnd
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
Wnd ∈ [W0 ... W15]

Operation:
```
Max_Shift = 17
Temp = (Ws)
Shift = 1
While ( (Shift < Max_Shift) && !(Temp & 0x8000) )
   Temp = Temp << 1
   Shift = Shift + 1
If (Shift == Max_Shift)
   0 → (Wnd)
Else
   Shift → (Wnd)
```

Status Affected: C

Encoding:

1100	1111	1000	0ddd	dppp	ssss

Description: Finds the first occurrence of a '1' starting from the Most Significant bit of Ws and working towards the Least Significant bit of the word operand. The bit number result is zero-extended to 16-bits and placed in Wnd.

Bit numbering begins with the Most Significant bit (allocated number 1) and advances to the Least Significant bit (allocated number 16). A result of zero indicates a '1' was not found, and the C flag will be set. If a '1' is found, the C flag is cleared.

The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: 1

Example 1
```
FF1L  W2, W5     ; Find the 1st one from the left in W2
                 ; and store result to W5
```

	Before Instruction		After Instruction
W2	000A	W2	000A
W5	BBBB	W5	000D
SR	0000	SR	0000

Example 2

```
FF1L  [W2++], W5  ; Find the 1st one from the left in [W2]
                  ; and store the result to W5
                  ; Post-increment W2
```

	Before Instruction		After Instruction	
W2	2000	W2	2002	
W5	BBBB	W5	0000	
Data 2000	0000	Data 2000	0000	
SR	0000	SR	0001	(C=1)

FF1R — Find First One from Right

Syntax:	{label:}	FF1R	Ws,	Wnd
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
Wnd ∈ [W0 ... W15]

Operation:
```
Max_Shift = 17
Temp = (Ws)
Shift = 1
While ( (Shift < Max_Shift) && !(Temp & 0x1) )
   Temp = Temp >> 1
   Shift = Shift + 1
If (Shift == Max_Shift)
   0 → (Wnd)
Else
   Shift → (Wnd)
```

Status Affected: C

Encoding:

1100	1111	0000	0ddd	dppp	ssss

Description: Finds the first occurrence of a '1' starting from the Least Significant bit of Ws and working towards the Most Significant bit of the word operand. The bit number result is zero-extended to 16-bits and placed in Wnd.

Bit numbering begins with the Least Significant bit (allocated number 1) and advances to the Most Significant bit (allocated number 16). A result of zero indicates a '1' was not found, and the C flag will be set. If a '1' is found, the C flag is cleared.

The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: 1

Example 1

```
FF1R  W1, W9   ; Find the 1st one from the right in W1
               ; and store the result to W9
```

	Before Instruction	After Instruction
W1	000A	000A
W9	BBBB	0002
SR	0000	0000

Example 2

```
FF1R  [W1++], W9  ; Find the 1st one from the right in [W1]
                  ; and store the result to W9
                  ; Post-increment W1
```

	Before Instruction		After Instruction
W1	2000	W1	2002
W9	BBBB	W9	0010
Data 2000	8000	Data 2000	8000
SR	0000	SR	0000

GOTO — Unconditional Jump

Syntax:	{label:}	GOTO	Expr

Operands: Expr may be label or expression (but not a literal).
Expr is resolved by the linker to a lit23, where lit23 ∈ [0 ... 8388606].

Operation: lit23 → PC
NOP → Instruction Register

Status Affected: None

Encoding:

1st word	0000	0100	nnnn	nnnn	nnnn	nnn0
2nd word	0000	0000	0000	0000	0nnn	nnnn

Description: Unconditional jump to anywhere within the 4M instruction word program memory range. The PC is loaded with the 23-bit literal specified in the instruction. Since the PC must always reside on an even address boundary, lit23<0> is ignored.

The 'n' bits form the target address.

Note: The linker will resolve the specified expression into the lit23 to be used.

Words: 2

Cycles: 2

Example 1

```
026000          GOTO   _THERE          ; Jump to _THERE
026004          MOV    W0, W1
   .            ...
   .            ...
027844 _THERE: MOV    #0x400, W2      ; Code execution
027846          ...                    ; resumes here
```

	Before Instruction		After Instruction
PC	02 6000	PC	02 7844
SR	0000	SR	0000

Example 2

```
000100 _code:  ...                     ; start of code
   .            ...
026000          GOTO   _code+2         ; Jump to _code+2
026004          ...
```

	Before Instruction		After Instruction
PC	02 6000	PC	00 0102
SR	0000	SR	0000

GOTO

Unconditional Indirect Jump

Syntax:	{label:}	GOTO	Wn

Operands: Wn ∈ [W0 ... W15]

Operation:
0 → PC<22:16>
(Wn<15:1>) → PC<15:1>
0 → PC<0>
NOP → Instruction Register

Status Affected: None

Encoding:

0000	0001	0100	0000	0000	ssss

Description: Unconditional indirect jump within the first 32K words of program memory. Zero is loaded into PC<22:16> and the value specified in (Wn) is loaded into PC<15:1>. Since the PC must always reside on an even address boundary, Wn<0> is ignored.

The ‘s’ bits select the address of the source register.

Words: 1

Cycles: 2

Example 1

```
006000          GOTO   W4              ; Jump unconditionally
006002          MOV    W0, W1          ; to 16-bit value in W4
   .            ...
   .            ...
007844 _THERE:  MOV   #0x400, W2       ; Code execution
007846          ...                    ; resumes here
```

	Before Instruction		After Instruction
W4	7844	W4	7844
PC	00 6000	PC	00 7844
SR	0000	SR	0000

5

Instruction Descriptions

INC

Increment f

Syntax:	{label:}	INC{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (f) + `1` → destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1100	0BDf	ffff	ffff	ffff

Description: Add one to the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

> **Note 1:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
>
> **2:** The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
INC.B  0x1000        ; Increment 0x1000 (Byte mode)
```

	Before Instruction		After Instruction	
Data 1000	8FFF	Data 1000	8F00	
SR	0000	SR	0101	(DC, C=1)

Example 2

```
INC   0x1000, WREG    ; Increment 0x1000 and store to WREG
                      ; (Word mode)
```

	Before Instruction		After Instruction	
WREG	ABCD	WREG	9000	
Data 1000	8FFF	Data 1000	8FFF	
SR	0000	SR	0108	(DC, N=1)

INC

Increment Ws

Syntax:	{label:}	INC{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Ws) + 1 → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1000	0Bqq	qddd	dppp	ssss

Description: Add one to the contents of the source register Ws and place the result in the destination register Wd. Register direct or indirect addressing may be used for Ws and Wd.

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
INC.B  W1, [++W2]        ; Pre-increment W2
                         ; Increment W1 and store to W2
                         ; (Byte mode)
```

	Before Instruction		After Instruction
W1	FF7F	W1	FF7F
W2	2000	W2	2001
Data 2000	ABCD	Data 2000	80CD
SR	0000	SR	010C (DC, N, OV=1)

Example 2

```
INC  W1, W2          ; Increment W1 and store to W2
                     ; (Word mode)
```

	Before Instruction		After Instruction
W1	FF7F	W1	FF7F
W2	2000	W2	FF80
SR	0000	SR	0108 (DC, N=1)

INC2

Increment f by 2

Syntax:	{label:}	INC2{.B}	f	{,WREG}

Operands: $f \in [0 \ldots 8191]$

Operation: (f) + 2 $\rightarrow$ destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1100	1BDf	ffff	ffff	ffff

Description: Add two to the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

> **Note:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
INC2.B  0x1000   ; Increment 0x1000 by 2
                 ; (Byte mode)
```

	Before Instruction		After Instruction	
Data 1000	8FFF	Data 1000	8F01	
SR	0000	SR	0101	(DC, C=1)

Example 2

```
INC2  0x1000, WREG  ; Increment 0x1000 by 2 and store to WREG
                    ; (Word mode)
```

	Before Instruction		After Instruction	
WREG	ABCD	WREG	9001	
Data 1000	8FFF	Data 1000	8FFF	
SR	0000	SR	0108	(DC, N=1)

INC2

Increment Ws by 2

Syntax:	{label:}	INC2{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Ws) + 2 → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1000	1Bqq	qddd	dppp	ssss

Description: Add two to the contents of the source register Ws and place the result in the destination register Wd. Register direct or indirect addressing may be used for Ws and Wd.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
INC2.B  W1, [++W2]    ; Pre-increment W2
                      ; Increment by 2 and store to W1
                      ; (Byte mode)
```

	Before Instruction		After Instruction	
W1	FF7F	W1	FF7F	
W2	2000	W2	2001	
Data 2000	ABCD	Data 2000	81CD	
SR	0000	SR	010C	(DC, N, OV=1)

Example 2

```
INC2  W1, W2      ; Increment W1 by 2 and store to W2
                  ; (word mode)
```

	Before Instruction		After Instruction	
W1	FF7F	W1	FF7F	
W2	2000	W2	FF81	
SR	0000	SR	0108	(DC, N=1)

IOR

Inclusive OR f and WREG

{label:}	IOR{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (f).IOR.(WREG) → destination designated by D

Status Affected: N, Z

Encoding:

1011	0111	0BDf	ffff	ffff	ffff

Description: Compute the logical inclusive OR operation of the contents of the working register WREG and the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
IOR.B  0x1000        ; IOR WREG to (0x1000) (Byte mode)
                     ; (Byte mode)
```

	Before Instruction		After Instruction
WREG	1234	WREG	1234
Data 1000	FF00	Data 1000	FF34
SR	0000	SR	0000

Example 2

```
IOR  0x1000, WREG    ; IOR (0x1000) to WREG
                     ; (Word mode)
```

	Before Instruction			After Instruction
WREG	1234		WREG	1FBF
Data 1000	0FAB		Data 1000	0FAB
SR	0008	(N=1)	SR	0000

IOR

Inclusive OR Literal and Wn

Syntax:	{label:} IOR{.B} #lit10, Wn
Operands:	lit10 ∈ [0 ... 255] for byte operation lit10 ∈ [0 ... 1023] for word operation Wn ∈ [W0 ... W15]
Operation:	lit10.IOR.(Wn) → Wn
Status Affected:	N, Z
Encoding:	`1011` `0011` `0Bkk` `kkkk` `kkkk` `dddd`

1011	0011	0Bkk	kkkk	kkkk	dddd

Description: Compute the logical inclusive OR operation of the 10-bit literal operand and the contents of the working register Wn and place the result back into the working register Wn.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘k’ bits specify the literal operand.
The ‘d’ bits select the address of the working register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See **Section 4.6 “Using 10-bit Literal Operands”** for information on using 10-bit literal operands in Byte mode.

Words: 1

Cycles: 1

Example 1

```
IOR.B #0xAA, W9        ; IOR 0xAA to W9
                       ; (Byte mode)
```

	Before Instruction		After Instruction	
W9	1234	W9	12BE	
SR	0000	SR	0008	(N=1)

Example 2

```
IOR  #0x2AA, W4        ; IOR 0x2AA to W4
                       ; (Word mode)
```

	Before Instruction		After Instruction	
W4	A34D	W4	A3EF	
SR	0000	SR	0008	(N=1)

IOR

Inclusive OR Wb and Short Literal

Syntax:	{label:}	IOR{.B}	Wb,	#lit5,	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[++Wd]
					[--Wd]

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
Wd ∈ [W0 ... W15]

Operation: (Wb).IOR.lit5 → Wd

Status Affected: N, Z

Encoding:

0111	0www	wBqq	qddd	d11k	kkkk

Description: Compute the logical inclusive OR operation of the contents of the base register Wb and the 5-bit literal operand and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘k’ bits provide the literal operand, a five-bit integer number.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
IOR.B  W1, #0x5, [W9++]   ; IOR W1 and 0x5 (Byte mode)
                          ; Store to [W9]
                          ; Post-increment W9
```

	Before Instruction		After Instruction	
W1	AAAA	W1	AAAA	
W9	2000	W9	2001	
Data 2000	0000	Data 2000	00AF	
SR	0000	SR	0008	(N=1)

Example 2

```
IOR  W1, #0x0, W9    ; IOR W1 with 0x0 (Word mode)
                     ; Store to W9
```

	Before Instruction		After Instruction	
W1	0000	W1	0000	
W9	A34D	W9	0000	
SR	0000	SR	0002	(Z=1)

 Advance Information

IOR

Inclusive OR Wb and Ws

Syntax:	{label:}	IOR{.B}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]
				[++Ws],	[++Wd]
				[--Ws],	[--Wd]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Wb).IOR.(Ws) → Wd

Status Affected: N, Z

Encoding:

0111	0www	wBqq	qddd	dppp	ssss

Description: Compute the logical inclusive OR operation of the contents of the source register Ws and the contents of the base register Wb and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
IOR.B  W1, [W5++], [W9++]  ; IOR W1 and [W5] (Byte mode)
                           ; Store result to [W9]
                           ; Post-increment W5 and W9
```

	Before Instruction	After Instruction	
W1	AAAA	AAAA	
W5	2000	2001	
W9	2400	2401	
Data 2000	1155	1155	
Data 2400	0000	00FF	
SR	0000	0008	(N=1)

5
Instruction Descriptions

Example 2

```
IOR  W1, W5, W9          ; IOR W1 and W5 (Word mode)
                         ; Store the result to W9
```

	Before Instruction		After Instruction	
W1	AAAA	W1	AAAA	
W5	5555	W5	5555	
W9	A34D	W9	FFFF	
SR	0000	SR	0008	(N=1)

LAC

Load Accumulator

Syntax:	{label:}	LAC	Ws,	{#Slit4,}	Acc
			[Ws],		
			[Ws++],		
			[Ws--],		
			[--Ws],		
			[++Ws],		
			[Ws+Wb],		

Operands:
Ws ∈ [W0 ... W15]
Wb ∈ [W0 ... W15]
Slit4 ∈ [-8 ... +7]
Acc ∈ [A,B]

Operation: $\text{Shift}_{\text{Slit4}}(\text{Extend(Ws)}) \rightarrow \text{Acc(A or B)}$

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1100	1010	Awww	wrrr	rggg	ssss

Description: Read the contents of the source register, optionally perform a signed 4-bit shift and store the result in the specified accumulator. The shift range is -8:7, where a negative operand indicates an arithmetic left shift and a positive operand indicates an arithmetic right shift. The data stored in the source register is assumed to be 1.15 fractional data and is automatically sign-extended (through bit 39) and zero-backfilled (bits [15:0]), prior to shifting.

The ‘A’ bit specifies the destination accumulator.
The ‘w’ bits specify the offset register Wb.
The ‘r’ bits encode the accumulator pre-shift.
The ‘g’ bits select the source Address mode.
The ‘s’ bits specify the source register Ws.

Note: If the operation moves more than sign-extension data into the upper Accumulator register (AccxU), or causes a saturation, the appropriate overflow and saturation bits will be set.

Words: 1

Cycles: 1

Example 1

```
LAC   [W4++], #-3, B     ; Load ACCB with [W4] << 3
                         ; Contents of [W4] do not change
                         ; Post increment W4
                         ; Assume saturation disabled
                         ; (SATB = 0)
```

	Before Instruction		After Instruction	
W4	2000	W4	2002	
ACCB	00 5125 ABCD	ACCB	FF 9108 0000	
Data 2000	1221	Data 2000	1221	
SR	0000	SR	4800	(OB, OAB=1)

5

Instruction Descriptions

Example 2

```
LAC   [--W2], #7, A      ; Pre-decrement W2
                         ; Load ACCA with [W2] >> 7
                         ; Contents of [W2] do not change
                         ; Assume saturation disabled
                         ; (SATA = 0)
```

	Before Instruction		After Instruction
W2	4002	W2	4000
ACCA	00 5125 ABCD	ACCA	FF FF22 1000
Data 4000	9108	Data 4000	9108
Data 4002	1221	Data 4002	1221
SR	0000	SR	0000

LNK

Allocate Stack Frame

Syntax:	{label:}	LNK	#lit14			
Operands:	lit14 ∈ [0 ... 16382]					
Operation:	(W14) → (TOS) (W15) + 2 → W15 (W15) → W14 (W15) + lit14 → W15					
Status Affected:	None					
Encoding:	1111	1010	00kk	kkkk	kkkk	kkk0

Description: This instruction allocates a stack frame of size lit14 bytes for a subroutine calling sequence. The stack frame is allocated by pushing the contents of the frame pointer (W14) onto the stack, storing the updated stack pointer (W15) to the frame pointer and then incrementing the stack pointer by the unsigned 14-bit literal operand. This instruction supports a maximum stack frame of 16382 bytes.

The ‘k’ bits specify the size of the stack frame.

Note: Since the stack pointer can only reside on a word boundary, lit14 must be even.

Words: 1

Cycles: 1

Example 1

```
LNK  #0xA0   ; Allocate a stack frame of 160 bytes
```

	Before Instruction		After Instruction
W14	2000	W14	2002
W15	2000	W15	20A2
Data 2000	0000	Data 2000	2000
SR	0000	SR	0000

LSR

Logical Shift Right f

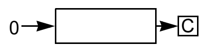
Syntax:	{label:}	LSR{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation:

For byte operation:
- 0 → Dest<7>
- (f<7:1>) → Dest<6:0>
- (f<0>) → C

For word operation:
- 0 → Dest<15>
- (f<15:1>) → Dest<14:0>
- (f<0>) → C



Status Affected: N, Z, C

Encoding:

1101	0101	0BDf	ffff	ffff	ffff

Description: Shift the contents of the file register one bit to the right and place the result in the destination register. The Least Significant bit of the file register is shifted into the Carry bit of the Status register. Zero is shifted into the Most Significant bit of the destination register.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
LSR.B  0x600    ; Logically shift right (0x600) by one
                ; (Byte mode)
```

	Before Instruction		After Instruction	
Data 600	55FF	Data 600	557F	
SR	0000	SR	0001	(C=1)

Example 2

```
LSR  0x600, WREG  ; Logically shift right (0x600) by one
                  ; Store to WREG
                  ; (Word mode)
```

	Before Instruction		After Instruction	
Data 600	55FF	Data 600	55FF	
WREG	0000	WREG	2AFF	
SR	0000	SR	0001	(C=1)

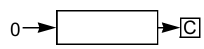
LSR

Logical Shift Right Ws

Syntax:	{label:}	LSR{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: For byte operation:
0 → Wd<7>
(Ws<7:1>) → Wd<6:0>
(Ws<0>) → C
For word operation:
0 → Wd<15>
(Ws<15:1>) → Wd<14:0>
(Ws<0>) → C



Status Affected: N, Z, C

Encoding:

1101	0001	0Bqq	qddd	dppp	ssss

Description: Shift the contents of the source register Ws one bit to the right and place the result in the destination register Wd. The Least Significant bit of Ws is shifted into the Carry bit of the Status register. Zero is shifted into the Most Significant bit of Wd. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘B’ bit selects byte or word operation (0 for word, 1 for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
LSR.B  W0, W1      ; LSR W0 (Byte mode)
                   ; Store result to W1
```

	Before Instruction		After Instruction	
W0	FF03	W0	FF03	
W1	2378	W1	2301	
SR	0000	SR	0001	(C=1)

5
Instruction Descriptions

Example 2

```
LSR  W0, W1        ; LSR W0 (Word mode)
                   ; Store the result to W1
```

	Before Instruction		After Instruction
W0	8000	W0	8000
W1	2378	W1	4000
SR	0000	SR	0000

LSR

Logical Shift Right by Short Literal

Syntax:	{label:}	LSR	Wb,	#lit4,	Wnd

Operands:
Wb ∈ [W0 ... W15]
lit4 ∈ [0 ... 15]
Wnd ∈ [W0 ... W15]

Operation:
lit4<3:0> → Shift_Val
0 → Wnd<15:15-Shift_Val+1>
Wb<15:Shift_Val> → Wnd<15-Shift_Val:0>

Status Affected: N, Z

Encoding:

1101	1110	0www	wddd	d100	kkkk

Description: Logical shift right the contents of the source register Wb by the 4-bit unsigned literal and store the result in the destination register Wnd. Direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the destination register.
The 'k' bits provide the literal operand.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: 1

Example 1

```
LSR    W4, #14, W5    ; LSR W4 by 14
                      ; Store result to W5
```

	Before Instruction	After Instruction
W4	C800	C800
W5	1200	0003
SR	0000	0000

Example 2

```
LSR    W4, #1, W5     ; LSR W4 by 1
                      ; Store result to W5
```

	Before Instruction	After Instruction
W4	0505	0505
W5	F000	0282
SR	0000	0000

LSR

Logical Shift Right by Wns

Syntax:	{label:}	LSR	Wb,	Wns,	Wnd

Operands: Wb ∈ [W0 ... W15]
Wns ∈ [W0 ...W15]
Wnd ∈ [W0 ... W15]

Operation: Wns<4:0> → Shift_Val
0 → Wnd<15:15-Shift_Val+1>
Wb<15:Shift_Val> → Wnd<15-Shift_Val:0>

Status Affected: N, Z

Encoding:

1101	1110	0www	wddd	d000	ssss

Description: Logical shift right the contents of the source register Wb by the 5 Least Significant bits of Wns (only up to 15 positions) and store the result in the destination register Wnd. Direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the destination register.
The 's' bits select the address of the source register.

Note 1: This instruction operates in Word mode only.
2: If Wns is greater than 15, Wnd will be loaded with `0x0`.

Words: 1

Cycles: 1

Example 1

```
LSR    W0, W1, W2     ; LSR W0 by W1
                      ; Store result to W2
```

	Before Instruction	After Instruction
W0	C00C	C00C
W1	0001	0001
W2	2390	6006
SR	0000	0000

Example 2

```
LSR    W5, W4, W3     ; LSR W5 by W4
                      ; Store result to W3
```

	Before Instruction	After Instruction
W3	DD43	0000
W4	000C	000C
W5	0800	0800
SR	0000	0002 (Z=1)

 Advance Information

MAC

Multiply and Accumulate

Syntax:	{label:}	MAC	Wm*Wn, Acc	{,[Wx], Wxd}	{,[Wy], Wyd}	{,AWB}
				{,[Wx]+=kx, Wxd}	{,[Wy]+=ky, Wyd}	
				{,[Wx]-=kx, Wxd}	{,[Wy]-=ky, Wyd}	
				{,[W9+W12], Wxd}	{,[W11+W12], Wyd}	

Operands:
Wm*Wn ∈ [W4*W5, W4*W6, W4*W7, W5*W6, W5*W7, W6*W7]
Acc ∈ [A,B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]
AWB ∈ [W13, [W13]+=2]

Operation:
(Acc(A or B)) + (Wm)*(Wn) → Acc(A or B)
([Wx])→ Wxd; (Wx)+kx→Wx
([Wy])→ Wyd; (Wy)+ky→Wy
(Acc(B or A)) rounded → AWB

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1100	0mmm	A0xx	yyii	iijj	jjaa

Description:
Multiply the contents of two working registers, optionally pre-fetch operands in preparation for another `MAC` type instruction and optionally store the unspecified accumulator results. The 32-bit result of the signed multiply is sign-extended to 40-bits and added to the specified accumulator.

Operands Wx, Wxd, Wy and Wyd specify optional pre-fetch operations, which support indirect and register offset addressing, as described in **Section 4.14.1 “MAC Pre-Fetches”**. Operand AWB specifies the optional store of the “other” accumulator, as described in **Section 4.14.4 “MAC Write Back”**.

The ‘m’ bits select the operand registers Wm and Wn for the multiply.
The ‘A’ bit selects the accumulator for the result.
The ‘x’ bits select the pre-fetch Wxd destination.
The ‘y’ bits select the pre-fetch Wyd destination.
The ‘i’ bits select the Wx pre-fetch operation.
The ‘j’ bits select the Wy pre-fetch operation.
The ‘a’ bits select the accumulator write back destination.

Note: The IF bit, CORCON<0>, determines if the multiply is fractional or an integer.

Words: 1

Cycles: 1

5

Instruction Descriptions

Example 1

```
MAC  W4*W5, A, [W8]+=6, W4, [W10]+=2, W5
; Multiply W4*W5 and add to ACCA
; Fetch [W8] to W4, Post-increment W8 by 6
; Fetch [W10] to W5, Post-increment W10 by 2
; CORCON = 0x00C0 (fractional multiply, normal saturation)
```

	Before Instruction	After Instruction
W4	A022	2567
W5	B900	909C
W8	0A00	0A06
W10	1800	1802
ACCA	00 1200 0000	00 472D 2400
Data 0A00	2567	2567
Data 1800	909C	909C
CORCON	00C0	00C0
SR	0000	0000

Example 2

```
MAC  W4*W5, A, [W8]-=2, W4, [W10]+=2, W5, W13
; Multiply W4*W5 and add to ACCA
; Fetch [W8] to W4, Post-decrement W8 by 2
; Fetch [W10] to W5, Post-increment W10 by 2
; Write Back ACCB to W13
; CORCON = 0x00D0 (fractional multiply, super saturation)
```

	Before Instruction	After Instruction
W4	1000	5BBE
W5	3000	C967
W8	0A00	09FE
W10	1800	1802
W13	2000	0001
ACCA	23 5000 2000	23 5600 2000
ACCB	00 0000 8F4C	00 0000 1F4C
Data 0A00	5BBE	5BBE
Data 1800	C967	C967
CORCON	00D0	00D0
SR	0000	8800 (OA, OAB=1)

MAC

Square and Accumulate

Syntax:	{label:}	MAC	Wm*Wm, Acc	{,[Wx], Wxd}	{,[Wy], Wyd}
				{,[Wx]+=kx, Wxd}	{,[Wy]+=ky, Wyd}
				{,[Wx]-=kx, Wxd}	{,[Wy]-=ky, Wyd}
				{,[W9+W12], Wxd}	{,[W11+W12], Wyd}

Operands: Wm*Wm ∈ [W4*W4, W5*W5, W6*W6, W7*W7]
Acc ∈ [A,B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]

Operation: (Acc(A or B)) + (Wm)*(Wm) → Acc(A or B)
([Wx])→ Wxd; (Wx)+kx→Wx
([Wy])→ Wyd; (Wy)+ky→Wy

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

`1111`	`00mm`	`A0xx`	`yyii`	`iijj`	`jj00`

Description: Square the contents of a working register, optionally pre-fetch operands in preparation for another `MAC` type instruction and optionally store the unspecified accumulator results. The 32-bit result of the signed multiply is sign-extended to 40-bits and added to the specified accumulator.

Operands Wx, Wxd, Wy and Wyd specify optional pre-fetch operations, which support indirect and register offset addressing, as described in **Section 4.14.1 “MAC Pre-Fetches”**.

The ‘m’ bits select the operand register Wm for the square.
The ‘A’ bit selects the accumulator for the result.
The ‘x’ bits select the pre-fetch Wxd destination.
The ‘y’ bits select the pre-fetch Wyd destination.
The ‘i’ bits select the Wx pre-fetch operation.
The ‘j’ bits select the Wy pre-fetch operation.

Note: The IF bit, CORCON<0>, determines if the multiply is fractional or an integer.

Words: 1

Cycles: 1

Example 1

```
MAC  W4*W4, B, [W9+W12], W4, [W10]-=2, W5
; Square W4 and add to ACCB
; Fetch [W9+W12] to W4
; Fetch [W10] to W5, Post-decrement W10 by 2
; CORCON = 0x00C0 (fractional multiply, normal saturation)
```

	Before Instruction	After Instruction
W4	A022	A230
W5	B200	650B
W9	0C00	0C00
W10	1900	18FE
W12	0020	0020
ACCB	00 2000 0000	00 67CD 0908
Data 0C20	A230	A230
Data 1900	650B	650B
CORCON	00C0	00C0
SR	0000	0000

Example 2

```
MAC  W7*W7, A, [W11]-=2, W7
; Square W7 and add to ACCA
; Fetch [W11] to W7, Post-decrement W11 by 2
; CORCON = 0x00D0 (fractional multiply, super saturation)
```

	Before Instruction	After Instruction
W7	76AE	23FF
W11	2000	1FFE
ACCA	FE 9834 4500	FF 063E 0188
Data 2000	23FF	23FF
CORCON	00D0	00D0
SR	0000	8800 (OA, OAB=1)

 Advance Information

MOV — Move f to Destination

Syntax:	{label:}	MOV{.B}	f	{,WREG}	

Operands: f ∈ [0 ... 8191]

Operation: (f) → destination designated by D

Status Affected: N, Z

Encoding:

1011	1111	1BDf	ffff	ffff	ffff

Description: Move the contents of the specified file register to the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored back to the file register and the only effect is to modify the status register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
MOV.B  TMR0, WREG  ; move (TMR0) to WREG (Byte mode)
```

	Before Instruction		After Instruction
WREG (W0)	9080	WREG (W0)	9055
TMR0	2355	TMR0	2355
SR	0000	SR	0000

Example 2

```
MOV  0x800         ; update SR based on (0x800) (Word mode)
```

	Before Instruction		After Instruction
Data 0800	B29F	Data 0800	B29F
SR	0000	SR	0008 (N=1)

5

Instruction Descriptions

MOV

Move WREG to f

Syntax:	{label:}	MOV{.B}	WREG,	f		
Operands:	f ∈ [0 ... 8191]					
Operation:	(WREG) → f					
Status Affected:	None					
Encoding:	1011	0111	1B1f	ffff	ffff	ffff

Description: Move the contents of the default working register WREG into the specified file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte move rather than a word move. You may use a `.W` extension to denote a word move, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
MOV.B  WREG, 0x801     ; move WREG to 0x801 (Byte mode)
```

	Before Instruction		After Instruction	
WREG (W0)	98F3	WREG (W0)	98F3	
Data 0800	4509	Data 0800	F309	
SR	0000	SR	0008	(N=1)

Example 2

```
MOV    WREG, DISICNT    ; move WREG to DISICNT
```

	Before Instruction		After Instruction
WREG (W0)	00A0	WREG (W0)	00A0
DISICNT	0000	DISICNT	00A0
SR	0000	SR	0000

 Advance Information

MOV

Move f to Wnd

Syntax:	{label:}	MOV	f,	Wnd

Operands: f ∈ [0 ... 65534]
Wnd ∈ [W0 ... W15]

Operation: (f) → Wnd

Status Affected: None

Encoding:

1000	0fff	ffff	ffff	ffff	dddd

Description: Move the word contents of the specified file register to Wnd. The file register may reside anywhere in the 32K words of data memory, but must be word aligned. Register direct addressing must be used for Wnd.

The ‘f’ bits select the address of the file register.
The ‘d’ bits select the address of the destination register.

Note 1: This instruction only operates on word operands.
2: Since the file register address must be word aligned, only the upper 15 bits of the file register address are encoded (bit 0 is assumed to be ‘0’).

Words: 1

Cycles: 1

Example 1 `MOV CORCON, W12 ; move CORCON to W12`

	Before Instruction		After Instruction
W12	78FA	W12	00F0
CORCON	00F0	CORCON	00F0
SR	0000	SR	0000

Example 2 `MOV 0x27FE, W3 ; move (0x27FE) to W3`

	Before Instruction		After Instruction
W3	0035	W3	ABCD
Data 27FE	ABCD	Data 27FE	ABCD
SR	0000	SR	0000

MOV — Move Wns to f

Syntax:	{label:}	MOV	Wns,	f

Operands: $f \in [0 \ldots 65534]$
$Wns \in [W0 \ldots W15]$

Operation: $(Wns) \rightarrow f$

Status Affected: None

Encoding:

1000	1fff	ffff	ffff	ffff	ssss

Description: Move the word contents of the working register Wns to the specified file register. The file register may reside anywhere in the 32K words of data memory, but must be word aligned. Register direct addressing must be used for Wn.

The 'f' bits select the address of the file register.
The 's' bits select the address of the source register.

Note 1: This instruction only operates on word operands.
2: Since the file register address must be word aligned, only the upper 15 bits of the file register address are encoded (bit `0` is assumed to be '`0`').

Words: 1

Cycles: 1

Example 1

```
MOV   W4, XMDOSRT     ; move W4 to XMODSRT
```

	Before Instruction		After Instruction
W4	1200	W4	1200
XMODSRT	1340	XMODSRT	1200
SR	0000	SR	0000

Example 2

```
MOV    W8, 0x1222     ; move W8 to data address 0x1222
```

	Before Instruction		After Instruction
W8	F200	W8	F200
Data 1222	FD88	Data 1222	F200
SR	0000	SR	0000

 Advance Information

MOV.B

Move 8-bit Literal to Wnd

Syntax:	{label:}	MOV.B	#lit8,	Wnd	

Operands: lit8 ∈ [0 ... 255]
Wnd ∈ [W0 ... W15]

Operation: lit8 → Wnd

Status Affected: None

Encoding:

1011	0011	1100	kkkk	kkkk	dddd

Description: The unsigned 8-bit literal ‘k’ is loaded into the lower byte of Wnd. The upper byte of Wnd is not changed. Register direct addressing must be used for Wnd.

The ‘k’ bits specify the value of the literal.
The ‘d’ bits select the address of the working register.

Note: This instruction operates in Byte mode and the `.B` extension must be provided.

Words: 1

Cycles: 1

Example 1

```
MOV.B   #0x17, W5     ; load W5 with #0x17 (Byte mode)
```

	Before Instruction		After Instruction
W5	7899	W5	7817
SR	0000	SR	0000

Example 2

```
MOV.B   #0xFE, W9     ; load W9 with #0xFE (Byte mode)
```

	Before Instruction		After Instruction
W9	AB23	W9	ABFE
SR	0000	SR	0000

MOV

Move 16-bit Literal to Wnd

Syntax:	{label:}	MOV	#lit16,	Wnd	

Operands: lit16 ∈ [-32768 ... 65535]
Wnd ∈ [W0 ... W15]

Operation: lit16 → Wnd

Status Affected: None

Encoding:

0010	kkkk	kkkk	kkkk	kkkk	dddd

Description: The 16-bit literal 'k' is loaded into Wnd. Register direct addressing must be used for Wnd.

The 'k' bits specify the value of the literal.
The 'd' bits select the address of the working register.

Note 1: This instruction operates only in Word mode.
2: The literal may be specified as a signed value [-32768:32767], or unsigned value [0:65535].

Words: 1

Cycles: 1

Example 1

```
MOV   #0x4231, W13     ; load W13 with #0x4231
```

	Before Instruction		After Instruction
W13	091B	W13	4231
SR	0000	SR	0000

Example 2

```
MOV   #0x4, W2         ; load W2 with #0x4
```

	Before Instruction		After Instruction
W2	B004	W2	0004
SR	0000	SR	0000

Example 3

```
MOV   #-1000, W8       ; load W8 with #-1000
```

	Before Instruction		After Instruction
W8	23FF	W8	FC18
SR	0000	SR	0000

 Advance Information

MOV

Move [Ws with offset] to Wnd

Syntax:	{label:}	MOV{.B}	[Ws+Slit10],	Wnd

Operands: Ws ∈ [W0 ... W15]
Slit10 ∈ [-512 ... 511] for byte operation
Slit10 ∈ [-1024 ... 1022] (even only) for word operation
Wnd ∈ [W0 ... W15]

Operation: [Ws+Slit10] → Wnd

Status Affected: None

Encoding:

1001	0kkk	kBkk	kddd	dkkk	ssss

Description: The contents of [Ws+Slit10] are loaded into Wnd. In Word mode, the range of Slit10 is increased to [-1024 ... 1022] and Slit10 must be even to maintain word address alignment. Register indirect addressing must be used for the source, and direct addressing must be used for Wnd.

The 'k' bits specify the value of the literal.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'd' bits select the address of the destination register.
The 's' bits select the address of the source register.

Note 1: The extension `.B` in the instruction denotes a byte move rather than a word move. You may use a `.W` extension to denote a word move, but it is not required.

2: In Byte mode, the range of Slit10 is not reduced as specified in **Section 4.6 "Using 10-bit Literal Operands"**, since the literal represents an address offset from Ws.

Words: 1

Cycles: 1

Example 1

```
MOV.B   [W8+0x13], W10  ; load W10 with [W8+0x13]
                        ; (Byte mode)
```

	Before Instruction		After Instruction
W8	1008	W8	1008
W10	4009	W10	4033
Data 101A	3312	Data 101A	3312
SR	0000	SR	0000

Example 2

```
MOV    [W4+0x3E8], W2  ; load W2 with [W4+0x3E8]
                       ; (Word mode)
```

	Before Instruction		After Instruction
W2	9088	W2	5634
W4	0800	W4	0800
Data 0BE8	5634	Data 0BE8	5634
SR	0000	SR	0000

MOV

Move Wns to [Wd with offset]

Syntax:	{label:}	MOV{.B}	Wns,	[Wd+Slit10]

Operands: Wns ∈ [W0 ... W15]
Slit10 ∈ [-512 ... 511] in Byte mode
Slit10 ∈ [-1024 ... 1022] (even only) in Word mode
Wd ∈ [W0 ... W15]

Operation: (Wns) → [Wd+Slit10]

Status Affected: None

Encoding:

1001	1kkk	kBkk	kddd	dkkk	ssss

Description: The contents of Wns are stored to [Wd+Slit10]. In Word mode, the range of Slit10 is increased to [-1024 ... 1022] and Slit10 must be even to maintain word address alignment. Register direct addressing must be used for Wns, and indirect addressing must be used for the destination.

The 'k' bits specify the value of the literal.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'd' bits select the address of the destination register.
The 's' bits select the address of the destination register.

Note 1: The extension `.B` in the instruction denotes a byte move rather than a word move. You may use a `.W` extension to denote a word move, but it is not required.

2: In Byte mode, the range of Slit10 is not reduced as specified in **Section 4.6 "Using 10-bit Literal Operands"**, since the literal represents an address offset from Wd.

Words: 1

Cycles: 1

Example 1

```
MOV.B  W0, [W1+0x7]     ; store W0 to [W1+0x7]
                        ; (Byte mode)
```

	Before Instruction		After Instruction
W0	9015	W0	9015
W1	1800	W1	1800
Data 1806	2345	Data 1806	1545
SR	0000	SR	0000

Example 2

```
MOV  W11, [W1-0x400]    ; store W11 to [W1-0x400]
                        ; (Word mode)
```

	Before Instruction		After Instruction
W1	1000	W1	1000
W11	8813	W11	8813
Data 0C00	FFEA	Data 0C00	8813
SR	0000	SR	0000

 Advance Information

MOV

Move Ws to Wd

Syntax:	{label:}	MOV{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[--Ws],	[--Wd]
			[++Ws],	[++Wd]
			[Ws+Wb],	[Wd+Wb]

Operands: Ws ∈ [W0 ... W15]
Wb ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Ws) → Wd

Status Affected: None

Encoding:

0111	1www	wBhh	hddd	dggg	ssss

Description: Move the contents of the source register into the destination register. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits define the offset register Wb.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘h’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘g’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note 1: The extension `.B` in the instruction denotes a byte move rather than a word move. You may use a `.W` extension to denote a word move, but it is not required.

2: When Register Offset Addressing mode is used for both the source and destination, the offset must be the same because the ‘w’ encoding bits are shared by Ws and Wd.

3: The instruction “`PUSH Ws`” translates to `MOV Ws, [W15++]`.

4: The instruction “`POP Wd`” translates to `MOV [--W15], Wd`.

Words: 1

Cycles: 1

Example 1

```
MOV.B   [W0--], W4  ; Move [W0] to W4 (Byte mode)
                    ; Post-decrement W0
```

	Before Instruction		After Instruction
W0	0A01	W0	0A00
W4	2976	W4	2989
Data 0A00	8988	Data 0A00	8988
SR	0000	SR	0000

Example 2

```
MOV   [W6++], [W2+W3]    ; Move [W6] to [W2+W3] (Word mode)
                         ; Post-increment W6
```

	Before Instruction		After Instruction
W2	0800	W2	0800
W3	0040	W3	0040
W6	1228	W6	122A
Data 0840	9870	Data 0840	0690
Data 1228	0690	Data 1228	0690
SR	0000	SR	0000

Advance Information

MOV.D

Double-Word Move from Source to Wnd

Syntax:	{label:}	MOV.D	Wns,	Wnd
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Wns ∈ [W0, W2, W4 ... W14]
Ws ∈ [W0 ... W15]
Wnd ∈ [W0, W2, W4 ... W14]

Operation: For direct addressing of source:
Wns → Wnd
Wns+1 → Wnd+1
For indirect addressing of source:
See Description

Status Affected: None

Encoding:

1011	1110	0000	0ddd	0ppp	ssss

Description: Move the double-word specified by the source to a destination working register pair (Wnd:Wnd+1). If register direct addressing is used for the source, the contents of two successive working registers (Wns:Wns+1) are moved to Wnd:Wnd+1. If indirect addressing is used for the source, Ws specifies the effective address for the Least Significant Word of the double-word. Any pre/post-increment or pre/post-decrement will adjust Ws by 4 bytes to accommodate for the double-word.

The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the first source register.

Note 1: This instruction only operates on double-words. See Figure 4-2 for information on how double-words are aligned in memory.
2: Wnd must be an even working register.
3: The instruction "`POP.D Wnd`" translates to `MOV.D [--W15], Wnd`.

Words: 1

Cycles: 2

Example 1

```
MOV.D   W2, W6    ; Move W2 to W6 (Double mode)
```

	Before Instruction		After Instruction
W2	12FB	W2	12FB
W3	9877	W3	9877
W6	9833	W6	12FB
W7	FCC6	W7	9877
SR	0000	SR	0000

Example 2

```
MOV.D  [W7--], W4   ; Move [W7] to W4 (Double mode)
                    ; Post-decrement W7
```

	Before Instruction	After Instruction
W4	B012	A319
W5	FD89	9927
W7	0900	08FC
Data 0900	A319	A319
Data 0902	9927	9927
SR	0000	0000

MOV.D

Double-Word Move from Wns to Destination

Syntax:	{label:}	MOV.D	Wns,	Wnd
				[Wd]
				[Wd++]
				[Wd--]
				[++Wd]
				[--Wd]

Operands: Wns ∈ [W0, W2, W4 ... W14]
Wnd ∈ [W0, W2, W4 ... W14]
Wd ∈ [W0 ... W15]

Operation: For direct addressing of destination:
Wns → Wnd
Wns+1 → Wnd+1
For indirect addressing of destination:
See Description

Status Affected: None

Encoding:

1011	1110	10qq	qddd	d000	sss0

Description: Move a double-word (Wns:Wns+1) to the specified destination. If register direct addressing is used for the destination, the contents of Wns:Wns+1 are stored to Wnd:Wnd+1. If indirect addressing is used for the destination, Wd specifies the effective address for the Least Significant Word of the double-word. Any pre/post-increment or pre/post-decrement will adjust Wd by 4 bytes to accommodate for the double-word.

The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 's' bits select the address of the source register pair.

Note 1: This instruction only operates on double-words. See Figure 4-2 for information on how double-words are aligned in memory.
2: Wnd must be an even working register.
3: The instruction `PUSH.D Ws` translates to `MOV.D Wns, [W15++]`.

Words: 1

Cycles: 2

Example 1

```
MOV.D  W10, W0    ; Move W10 to W0 (Double mode)
```

	Before Instruction	After Instruction
W0	9000	CCFB
W1	4322	0091
W10	CCFB	CCFB
W11	0091	0091
SR	0000	0000

Example 2

```
MOV.D  W4, [--W6]  ; Pre-decrement W6 (Double mode)
                   ; Move W4 to [W6]
```

	Before Instruction		After Instruction
W4	100A	W4	100A
W5	CF12	W5	CF12
W6	0804	W6	0800
Data 0800	A319	Data 0800	100A
Data 0802	9927	Data 0802	CF12
SR	0000	SR	0000

MOVSAC **Pre-Fetch Operands and Store Accumulator**

Syntax:	{label:} MOVSAC	Acc	{,[Wx], Wxd}	{,[Wy], Wyd}	{,AWB}
			{,[Wx]+=kx, Wxd}	{,[Wy]+=ky, Wyd}	
			{,[Wx]-=kx, Wxd}	{,[Wy]-=ky, Wyd}	
			{,[W9+W12], Wxd}	{,[W11+W12], Wyd}	

Operands: Acc ∈ [A,B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]
AWB ∈ [W13, [W13]+=2]

Operation: ([Wx])→ Wxd; (Wx)+kx→Wx
([Wy])→ Wyd; (Wy)+ky→Wy
(Acc(B or A)) rounded → AWB

Status Affected: None

Encoding:

1100	0111	A0xx	yyii	iijj	jjaa

Description: Optionally pre-fetch operands in preparation for another MAC type instruction and optionally store the unspecified accumulator results. Even though an accumulator operation is not performed in this instruction, an accumulator must be specified to designate which accumulator to write back.

Operands Wx, Wxd, Wy and Wyd specify optional pre-fetch operations which support indirect and register offset addressing, as described in **Section 4.14.1 “MAC Pre-Fetches”**. Operand AWB specifies the optional store of the “other” accumulator, as described in **Section 4.14.4 “MAC Write Back”**.

The ‘A’ bit selects the other accumulator used for write back.
The ‘x’ bits select the pre-fetch Wxd destination.
The ‘y’ bits select the pre-fetch Wyd destination.
The ‘i’ bits select the Wx pre-fetch operation.
The ‘j’ bits select the Wy pre-fetch operation.
The ‘a’ bits select the accumulator write back destination.

Words: 1

Cycles: 1

Example 1

```
MOVSAC  B, [W9], W6, [W11]+=4, W7, W13
; Fetch [W9] to W6
; Fetch [W11] to W7, Post-increment W11 by 4
; Store ACCA to W13
```

	Before Instruction		After Instruction
W6	A022	W6	7811
W7	B200	W7	B2AF
W9	0800	W9	0800
W11	1900	W11	1904
W13	0020	W13	3290
ACCA	00 3290 5968	ACCA	00 3290 5968
Data 0800	7811	Data 0800	7811
Data 1900	B2AF	Data 1900	B2AF
SR	0000	SR	0000

Example 2

```
MOVSAC  A, [W9]-=2, W4, [W11+W12], W6, [W13]+=2
; Fetch [W9] to W4, Post-decrement W9 by 2
; Fetch [W11+W12] to W6
; Store ACCB to [W13], Post-increment W13 by 2
```

	Before Instruction	After Instruction
W4	76AE	BB00
W6	2000	52CE
W9	1200	11FE
W11	2000	2000
W12	0024	0024
W13	2300	2302
ACCB	00 9834 4500	00 9834 4500
Data 1200	BB00	BB00
Data 2024	52CE	52CE
Data 2300	23FF	9834
SR	0000	0000

MPY

Multiply Wm by Wn to Accumulator

Syntax:	{label:}	MPY	Wm*Wn, Acc	{,[Wx], Wxd}	{,[Wy], Wyd}
				{,[Wx]+=kx, Wxd}	{,[Wy]+=ky, Wyd}
				{,[Wx]-=kx, Wxd}	{,[Wy]-=ky, Wyd}
				{,[W9+W12], Wxd}	{,[W11+W12], Wyd}

Operands:
Wm*Wn ∈ [W4*W5, W4*W6, W4*W7, W5*W6, W5*W7, W6*W7]
Acc ∈ [A,B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]
AWB ∈ [W13, [W13]+=2]

Operation:
(Wm)*(Wn) → Acc(A or B)
([Wx])→ Wxd; (Wx)+kx→Wx
([Wy])→ Wyd; (Wy)+ky→Wy

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1100	0mmm	A0xx	yyii	iijj	jj11

Description: Multiply the contents of two working registers, optionally pre-fetch operands in preparation for another `MAC` type instruction and optionally store the unspecified accumulator results. The 32-bit result of the signed multiply is sign-extended to 40-bits and stored to the specified accumulator.

Operands Wx, Wxd, Wy and Wyd specify optional pre-fetch operations which support indirect and register offset addressing, as described in **Section 4.14.1 "MAC Pre-Fetches"**.

The 'm' bits select the operand registers Wm and Wn for the multiply:
The 'A' bit selects the accumulator for the result.
The 'x' bits select the pre-fetch Wxd destination.
The 'y' bits select the pre-fetch Wyd destination.
The 'i' bits select the Wx pre-fetch operation.
The 'j' bits select the Wy pre-fetch operation.

Note: The IF bit, CORCON<0>, determines if the multiply is fractional or an integer.

Words: 1

Cycles: 1

Example 1

```
MPY  W4*W5, A, [W8]+=2, W6, [W10]-=2, W7
; Multiply W4*W5 and store to ACCA
; Fetch [W8] to W6, Post-increment W8 by 2
; Fetch [W10] to W7, Post-decrement W10 by 2
; CORCON = 0x0000 (fractional multiply, no saturation)
```

	Before Instruction	After Instruction
W4	C000	C000
W5	9000	9000
W6	0800	671F
W7	B200	E3DC
W8	1780	1782
W10	2400	23FE
ACCA	FF F780 2087	00 3800 0000
Data 1780	671F	671F
Data 2400	E3DC	E3DC
CORCON	0000	0000
SR	0000	0000

Example 2

```
MPY  W6*W7, B, [W8]+=2, W4, [W10]-=2, W5
; Multiply W6*W7 and store to ACCB
; Fetch [W8] to W4, Post-increment W8 by 2
; Fetch [W10] to W5, Post-decrement W10 by 2
; CORCON = 0x0000 (fractional multiply, no saturation)
```

	Before Instruction	After Instruction
W4	C000	8FDC
W5	9000	0078
W6	671F	671F
W7	E3DC	E3DC
W8	1782	1784
W10	23FE	23FC
ACCB	00 9834 4500	FF E954 3748
Data 1782	8FDC	8FDC
Data 23FE	0078	0078
CORCON	0000	0000
SR	0000	0000

MPY

Square to Accumulator

Syntax:	{label:}	MPY	Wm*Wm, Acc	{,[Wx], Wxd}	{,[Wy], Wyd}
				{,[Wx]+=kx, Wxd}	{,[Wy]+=ky, Wyd}
				{,[Wx]-=kx, Wxd}	{,[Wy]-=ky, Wyd}
				{,[W9+W12], Wxd}	{,[W11+W12], Wyd}

Operands: Wm*Wm ∈ [W4*W4, W5*W5, W6*W6, W7*W7]
Acc ∈ [A,B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]

Operation: (Wm)*(Wm) → Acc(A or B)
([Wx])→ Wxd; (Wx)+kx→Wx
([Wy])→ Wyd; (Wy)+ky→Wy

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1111	00mm	A0xx	yyii	iijj	jj01

Description: Square the contents of a working register, optionally pre-fetch operands in preparation for another `MAC` type instruction and optionally store the unspecified accumulator results. The 32-bit result of the signed multiply is sign-extended to 40-bits and stored in the specified accumulator.

Operands Wx, Wxd, Wy and Wyd specify optional pre-fetch operations which support indirect and register offset addressing, as described in **Section 4.14.1 “MAC Pre-Fetches”**.

The ‘m’ bits select the operand register Wm for the square.
The ‘A’ bit selects the accumulator for the result.
The ‘x’ bits select the pre-fetch Wxd destination.
The ‘y’ bits select the pre-fetch Wyd destination.
The ‘i’ bits select the Wx pre-fetch operation.
The ‘j’ bits select the Wy pre-fetch operation.

Note: The IF bit, CORCON<0>, determines if the multiply is fractional or an integer.

Words: 1

Cycles: 1

Example 1

```
MPY  W6*W6, A, [W9]+=2, W6
; Square W6 and store to ACCA
; Fetch [W9] to W6, Post-increment W9 by 2
; CORCON = 0x0000 (fractional multiply, no saturation)
```

	Before Instruction	After Instruction
W6	6500	B865
W9	0900	0902
ACCA	00 7C80 0908	00 4FB2 0000
Data 0900	B865	B865
CORCON	0000	0000
SR	0000	0000

Example 2

```
MPY  W4*W4, B, [W9+W12], W4, [W10]+=2, W5
; Square W4 and store to ACCB
; Fetch [W9+W12] to W4
; Fetch [W10] to W5, Post-increment W10 by 2
; CORCON = 0x0000 (fractional multiply, no saturation)
```

	Before Instruction		After Instruction
W4	E228	W4	8911
W5	9000	W5	F678
W9	1700	W9	1700
W10	1B00	W10	1B02
W12	FF00	W12	FF00
ACCB	00 9834 4500	ACCB	00 06F5 4C80
Data 1600	8911	Data 1600	8911
Data 1B00	F678	Data 1B00	F678
CORCON	0000	CORCON	0000
SR	0000	SR	0000

MPY.N

Multiply -Wm by Wn to Accumulator

Syntax:	{label:}	MPY.N	Wm*Wn, Acc	{,[Wx], Wxd}	{,[Wy], Wyd}
				{,[Wx]+=kx, Wxd}	{,[Wy]+=ky, Wyd}
				{,[Wx]-=kx, Wxd}	{,[Wy]-=ky, Wyd}
				{,[W9+W12], Wxd}	{,[W11+W12], Wyd}

Operands:
Wm*Wn ∈ [W4*W5; W4*W6; W4*W7; W5*W6; W5*W7; W6*W7]
Acc ∈ [A,B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]

Operation:
-(Wm)*(Wn) → Acc(A or B)
([Wx])→ Wxd; (Wx)+kx→Wx
([Wy])→ Wyd; (Wy)+ky→Wy

Status Affected: OA, OB, OAB

Encoding:

1100	0mmm	A1xx	yyii	iijj	jj11

Description: Multiply the contents of a working register by the negative of the contents of another working register, optionally pre-fetch operands in preparation for another `MAC` type instruction and optionally store the unspecified accumulator results. The 32-bit result of the signed multiply is sign-extended to 40-bits and stored to the specified accumulator.

The ‘m’ bits select the operand registers Wm and Wn for the multiply.
The ‘A’ bit selects the accumulator for the result.
The ‘x’ bits select the pre-fetch Wxd destination.
The ‘y’ bits select the pre-fetch Wyd destination.
The ‘i’ bits select the Wx pre-fetch operation.
The ‘j’ bits select the Wy pre-fetch operation.

Note: The IF bit, CORCON<0>, determines if the multiply is fractional or an integer.

Words: 1

Cycles: 1

Example 1

```
MPY.N  W4*W5, A, [W8]+=2, W4, [W10]+=2, W5
; Multiply W4*W5, negate the result and store to ACCA
; Fetch [W8] to W4, Post-increment W8 by 2
; Fetch [W10] to W5, Post-increment W10 by 2
; CORCON = 0x0001 (integer multiply, no saturation)
```

	Before Instruction		After Instruction
W4	3023	W4	0054
W5	1290	W5	660A
W8	0B00	W8	0B02
W10	2000	W10	2002
ACCA	00 0000 2387	ACCA	FF FC82 7650
Data 0B00	0054	Data 0B00	0054
Data 2000	660A	Data 2000	660A
CORCON	0001	CORCON	0001
SR	0000	SR	0000

Example 2

```
MPY.N  W4*W5, A, [W8]+=2, W4, [W10]+=2, W5
; Multiply W4*W5, negate the result and store to ACCA
; Fetch [W8] to W4, Post-increment W8 by 2
; Fetch [W10] to W5, Post-increment W10 by 2
; CORCON = 0x0000 (fractional multiply, no saturation)
```

	Before Instruction	After Instruction
W4	3023	0054
W5	1290	660A
W8	0B00	0B02
W10	2000	2002
ACCA	00 0000 2387	FF F904 ECA0
Data 0B00	0054	0054
Data 2000	660A	660A
CORCON	0000	0000
SR	0000	0000

 Advance Information

MSC

Multiply and Subtract from Accumulator

Syntax:	{label:}	MSC	Wm*Wn, Acc	{,[Wx], Wxd}	{,[Wy], Wyd}	{,AWB}
				{,[Wx]+=kx, Wxd}	{,[Wy]+=ky, Wyd}	
				{,[Wx]-=kx, Wxd}	{,[Wy]-=ky, Wyd}	
				{,[W9+W12], Wxd}	{,[W11+W12], Wyd}	

Operands:
Wm*Wn ∈ [W4*W5, W4*W6, W4*W7, W5*W6, W5*W7, W6*W7]
Acc ∈ [A,B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]
AWB ∈ [W13, [W13]+=2]

Operation:
(Acc(A or B)) – (Wm)*(Wn) → Acc(A or B)
([Wx])→ Wxd; (Wx)+kx→Wx
([Wy])→ Wyd; (Wy)+ky→Wy
(Acc(B or A)) rounded → AWB

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1100	0mmm	A1xx	yyii	iijj	jjaa

Description: Multiply the contents of two working registers, optionally pre-fetch operands in preparation for another MAC type instruction and optionally store the unspecified accumulator results. The 32-bit result of the signed multiply is sign-extended to 40-bits and subtracted from the specified accumulator.

Operands Wx, Wxd, Wy and Wyd specify optional pre-fetch operations which support indirect and register offset addressing as described in **Section 4.14.1 "MAC Pre-Fetches"**. Operand AWB specifies the optional store of the "other" accumulator as described in **Section 4.14.4 "MAC Write Back"**.

The 'm' bits select the operand registers Wm and Wn for the multiply.
The 'A' bit selects the accumulator for the result.
The 'x' bits select the pre-fetch Wxd destination.
The 'y' bits select the pre-fetch Wyd destination.
The 'i' bits select the Wx pre-fetch operation.
The 'j' bits select the Wy pre-fetch operation.
The 'a' bits select the accumulator write back destination.

Note: The IF bit, CORCON<0>, determines if the multiply is fractional or an integer.

Words: 1

Cycles: 1

Example 1

```
MSC  W6*W7, A, [W8]-=4, W6, [W10]-=4, W7
; Multiply W6*W7 and subtract the result from ACCA
; Fetch [W8] to W6, Post-decrement W8 by 4
; Fetch [W10] to W7, Post-decrement W10 by 4
; CORCON = 0x0001 (integer multiply, no saturation)
```

	Before Instruction		After Instruction
W6	9051	W6	D309
W7	7230	W7	100B
W8	0C00	W8	0BFC
W10	1C00	W10	1BFC
ACCA	00 0567 8000	ACCA	00 3738 5ED0
Data 0C00	D309	Data 0C00	D309
Data 1C00	100B	Data 1C00	100B
CORCON	0001	CORCON	0001
SR	0000	SR	0000

Example 2

```
MSC  W4*W5, B, [W11+W12], W5, W13
; Multiply W4*W5 and subtract the result from ACCA
; Fetch [W11+W12] to W5
; Write Back ACCB to W13
; CORCON = 0x0000 (fractional multiply, no saturation)
```

	Before Instruction		After Instruction
W4	0500	W4	0500
W5	2000	W5	3579
W11	1800	W11	1800
W12	0800	W12	0800
W13	6233	W13	3738
ACCA	00 3738 5ED0	ACCA	00 3738 5ED0
ACCB	00 1000 0000	ACCB	00 0EC0 0000
Data 2000	3579	Data 2000	3579
CORCON	0000	CORCON	0000
SR	0000	SR	0000

MUL — Integer Unsigned Multiply f and WREG

Syntax:	{label:} MUL{.B} f
Operands:	f ∈ [0 ... 8191]
Operation:	For byte operation: (WREG)<7:0> * (f)<7:0> → W2 For word operation: (WREG) * (f) → W2:W3
Status Affected:	None
Encoding:	`1011` `1100` `0B0f` `ffff` `ffff` `ffff`

Description: Multiply the default working register WREG with the specified file register and place the result in the W2:W3 register pair. Both operands and the result are interpreted as unsigned integers. If this instruction is executed in Byte mode, the 16-bit result is stored in W2. In Word mode, the Most Significant Word of the 32-bit result is stored in W3, and the Least Significant Word of the 32-bit result is stored in W2.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The WREG is set to working register W0.
3: The IF bit, CORCON<0>, has no effect on this operation.
4: This is the only instruction which provides for an 8-bit multiply.

Words: 1

Cycles: 1

Example 1

```
MUL.B  0x800    ; Multiply (0x800)*WREG (Byte mode)
```

	Before Instruction	After Instruction
WREG (W0)	9823	9823
W2	FFFF	13B0
W3	FFFF	FFFF
Data 0800	2690	2690
SR	0000	0000

Example 2

```
MUL  TMR1       ; Multiply (TMR1)*WREG (Word mode)
```

	Before Instruction	After Instruction
WREG (W0)	F001	F001
W2	0000	C287
W3	0000	2F5E
TMR1	3287	3287
SR	0000	0000

MUL.SS

Integer 16x16-bit Signed Multiply

Syntax:	{label:}	MUL.SS	Wb,	Ws,	Wnd
				[Ws],	
				[Ws++],	
				[Ws--],	
				[++Ws],	
				[--Ws],	

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wnd ∈ [W0, W2, W4 ... W12]

Operation: signed (Wb) * signed (Ws) → Wnd:Wnd+1

Status Affected: None

Encoding:

1011	1001	1www	wddd	dppp	ssss

Description: Multiply the contents of Wb with the contents of Ws, and store the 32-bit result in two successive working registers. The Least Significant Word of the result is stored in Wnd (which must be an even numbered working register), and the Most Significant Word of the result is stored in Wnd+1. Both source operands and the result Wnd are interpreted as two's complement signed integers. Register direct addressing must be used for Wb and Wnd. Register direct or register indirect addressing may be used for Ws.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the lower destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note 1: This instruction operates in Word mode only.
2: Since the product of the multiplication is 32-bits, Wnd must be an even working register. See Figure 4-2 for information on how double-words are aligned in memory.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit, CORCON<0>, has no effect on this operation.

Words: 1

Cycles: 1

Example 1

```
MUL.SS  W0, W1, W12   ; Multiply W0*W1
                      ; Store the result to W12:W13
```

	Before Instruction	After Instruction
W0	9823	9823
W1	67DC	67DC
W12	FFFF	D314
W13	FFFF	D5DC
SR	0000	0000

Example 2

```
MUL.SS  W2, [--W4], W0   ; Pre-decrement W4
                         ; Multiply W2*[W4]
                         ; Store the result to W0:W1
```

	Before Instruction		After Instruction
W0	FFFF	W0	28F8
W1	FFFF	W1	0000
W2	0045	W2	0045
W4	27FE	W4	27FC
Data 27FC	0098	Data 27FC	0098
SR	0000	SR	0000

MUL.SU

Integer 16x16-bit Signed-Unsigned Short Literal Multiply

Syntax:	{label:} MUL.SU Wb, #lit5, Wnd
Operands:	Wb ∈ [W0 ... W15] lit5 ∈ [0 ... 31] Wnd ∈ [W0, W2, W4 ... W12]
Operation:	signed (Wb) * unsigned lit5 → Wnd:Wnd+1
Status Affected:	None
Encoding:	`1011` `1001` `0www` `wddd` `d11k` `kkkk`

Description: Multiply the contents of Wb with the 5-bit literal, and store the 32-bit result in two successive working registers. The Least Significant Word of the result is stored in Wnd (which must be an even numbered working register), and the Most Significant Word of the result is stored in Wnd+1. The Wb operand and the result Wnd are interpreted as a two's complement signed integer. The literal is interpreted as an unsigned integer. Register direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the lower destination register.
The 'k' bits define a 5-bit unsigned integer literal.

Note 1: This instruction operates in Word mode only.
2: Since the product of the multiplication is 32-bits, Wnd must be an even working register. See Figure 4-2 for information on how double-words are aligned in memory.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit, CORCON<0>, has no effect on this operation.

Words: 1

Cycles: 1

Example 1

```
MUL.SU  W0, #0x1F, W2  ; Multiply W0 by literal 0x1F
                       ; Store the result to W2:W3
```

	Before Instruction	After Instruction
W0	C000	C000
W2	1234	4000
W3	C9BA	FFF8
SR	0000	0000

Example 2

```
MUL.SU  W2, #0x10, W0  ; Multiply W2 by literal 0x10
                       ; Store the result to W0:W1
```

	Before Instruction		After Instruction
W0	ABCD	W0	2400
W1	89B3	W1	000F
W2	F240	W2	F240
SR	0000	SR	0000

MUL.SU

Integer 16x16-bit Signed-Unsigned Multiply

Syntax:	{label:}	MUL.SU	Wb,	Ws,	Wnd
				[Ws],	
				[Ws++],	
				[Ws--],	
				[++Ws],	
				[--Ws],	

Operands: $Wb \in$ [W0 ... W15]
$Ws \in$ [W0 ... W15]
$Wnd \in$ [W0, W2, W4 ... W12]

Operation: signed (Wb) * unsigned (Ws) $\rightarrow$ Wnd:Wnd+1

Status Affected: None

Encoding:

1011	1001	0www	wddd	dppp	ssss

Description: Multiply the contents of Wb with the contents of Ws, and store the 32-bit result in two successive working registers. The Least Significant Word of the result is stored in Wnd (which must be an even numbered working register), and the Most Significant Word of the result is stored in Wnd+1. The Wb operand and the result Wnd are interpreted as a two's complement signed integer. The Ws operand is interpreted as an unsigned integer. Register direct addressing must be used for Wb and Wnd. Register direct or register indirect addressing may be used for Ws.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the lower destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note 1: This instruction operates in Word mode only.
2: Since the product of the multiplication is 32-bits, Wnd must be an even working register. See Figure 4-2 for information on how double-words are aligned in memory.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit, CORCON<0>, has no effect on this operation.

Words: 1

Cycles: 1

Example 1

```
MUL.SU  W8, [W9], W0  ; Multiply W8*[W9]
                      ; Store the result to W0:W1
```

	Before Instruction		After Instruction
W0	68DC	W0	0000
W1	AA40	W1	F100
W8	F000	W8	F000
W9	178C	W9	178C
Data 178C	F000	Data 178C	F000
SR	0000	SR	0000

 Advance Information

Example 2

```
MUL.SU  W2, [++W3], W4  ; Pre-Increment W3
                        ; Multiply W2*[W3]
                        ; Store the result to W4:W5
```

	Before Instruction		After Instruction
W2	0040	W2	0040
W3	0280	W3	0282
W4	1819	W4	1A00
W5	2021	W5	0000
Data 0282	0068	Data 0282	0068
SR	0000	SR	0000

MUL.US

Integer 16x16-bit Unsigned-Signed Multiply

Syntax:	{label:}	MUL.US	Wb,	Ws,	Wnd
				[Ws],	
				[Ws++],	
				[Ws--],	
				[++Ws],	
				[--Ws],	

Operands:
Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wnd ∈ [W0, W2, W4 ... W12]

Operation: unsigned (Wb) * signed (Ws) → Wnd:Wnd+1

Status Affected: None

Encoding:

1011	1000	1www	wddd	dppp	ssss

Description: Multiply the contents of Wb with the contents of Ws, and store the 32-bit result in two successive working registers. The Least Significant Word of the result is stored in Wnd (which must be an even numbered working register), and the Most Significant Word of the result is stored in Wnd+1. The Wb operand is interpreted as an unsigned integer. The Ws operand and the result Wnd are interpreted as a two's complement signed integer. Register direct addressing must be used for Wb and Wnd. Register direct or register indirect addressing may be used for Ws.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the lower destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note 1: This instruction operates in Word mode only.
2: Since the product of the multiplication is 32-bits, Wnd must be an even working register. See Figure 4-2 for information on how double-words are aligned in memory.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit, CORCON<0>, has no effect on this operation.

Words: 1

Cycles: 1

Example 1

```
MUL.US  W0, [W1], W2  ; Multiply W0*[W1] (unsigned-signed)
                      ; Store the result to W2:W3
```

	Before Instruction	After Instruction
W0	C000	C000
W1	2300	2300
W2	00DA	0000
W3	CC25	F400
Data 2300	F000	F000
SR	0000	0000

Example 2

```
MUL.US  W6, [W5++], W10 ; Mult. W6*[W5] (unsigned-signed)
                        ; Store the result to W10:W11
                        ; Post-Increment W5
```

	Before Instruction		After Instruction
W5	0C00	W5	0C02
W6	FFFF	W6	FFFF
W10	0908	W10	8001
W11	6EEB	W11	7FFE
Data 0C00	7FFF	Data 0C00	7FFF
SR	0000	SR	0000

MUL.UU

Integer 16x16-bit Unsigned Short Literal Multiply

Syntax:	{label:}	MUL.UU	Wb,	#lit5,	Wnd

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
Wnd ∈ [W0, W2, W4 ... W12]

Operation: unsigned (Wb) * unsigned lit5 → Wnd:Wnd+1

Status Affected: None

Encoding:

1011	1000	0www	wddd	d11k	kkkk

Description: Multiply the contents of Wb with the 5-bit literal, and store the 32-bit result in two successive working registers. The Least Significant Word of the result is stored in Wnd (which must be an even numbered working register), and the Most Significant Word of the result is stored in Wnd+1. Both operands and the result are interpreted as unsigned integers. Register direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the lower destination register.
The 'k' bits define a 5-bit unsigned integer literal.

Note 1: This instruction operates in Word mode only.
2: Since the product of the multiplication is 32-bits, Wnd must be an even working register. See Figure 4-2 for information on how double-words are aligned in memory.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit, CORCON<0>, has no effect on this operation.

Words: 1

Cycles: 1

Example 1

```
MUL.UU  W0, #0xF, W12  ; Multiply W0 by literal 0xF
                       ; Store the result to W12:W13
```

	Before Instruction	After Instruction
W0	2323	2323
W12	4512	0F0D
W13	7821	0002
SR	0000	0000

Example 2

```
MUL.UU  W7, #0x1F, W0  ; Multiply W7 by literal 0x1F
                       ; Store the result to W0:W1
```

	Before Instruction	After Instruction
W0	780B	55C0
W1	3805	001D
W7	F240	F240
SR	0000	0000

 Advance Information

MUL.UU

Integer 16x16-bit Unsigned Multiply

Syntax:	{label:}	MUL.UU	Wb,	Ws,	Wnd
				[Ws],	
				[Ws++],	
				[Ws--],	
				[++Ws],	
				[--Ws],	

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wnd ∈ [W0, W2, W4 ... W12]

Operation: unsigned (Wb) * unsigned (Ws) → Wnd:Wnd+1

Status Affected: None

Encoding:

1011	1000	0www	wddd	dppp	ssss

Description: Multiply the contents of Wb with the contents of Ws, and store the 32-bit result in two successive working registers. The least Significant Word of the result is stored in Wnd (which must be an even numbered working register), and the most Significant Word of the result is stored in Wnd+1. Both source operands and the result are interpreted as unsigned integers. Register direct addressing must be used for Wb and Wnd. Register direct or indirect addressing may be used for Ws.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the lower destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note 1: This instruction operates in Word mode only.
2: Since the product of the multiplication is 32-bits, Wnd must be an even working register. See Figure 4-2 for information on how double-words are aligned in memory.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit, CORCON<0>, has no effect on this operation.

Words: 1

Cycles: 1

Example 1

```
MUL.UU  W4, W0, W2  ; Multiply W4*W0 (unsigned-unsigned)
                    ; Store the result to W2:W3
```

	Before Instruction	After Instruction
W0	FFFF	FFFF
W2	2300	0001
W3	00DA	FFFE
W4	FFFF	FFFF
SR	0000	0000

Example 2

```
MUL.UU W0, [W1++], W4  ; Mult. W0*[W1] (unsigned-unsigned)
                       ; Store the result to W4:W5
                       ; Post-Increment W1
```

	Before Instruction		After Instruction
W0	1024	W0	1024
W1	2300	W1	2302
W4	9654	W4	6D34
W5	BDBC	W5	0D80
Data 2300	D625	Data 2300	D625
SR	0000	SR	0000

NEG — Negate f

Syntax:	{label:}	NEG{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: $\overline{(f)}$ + 1 → destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1110	0BDf	ffff	ffff	ffff

Description: Compute the 2's complement of the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
NEG.B  0x880, WREG  ; Negate (0x880) (Byte mode)
                    ; Store result to WREG
```

	Before Instruction		After Instruction	
WREG (W0)	9080	WREG (W0)	90AB	
Data 0880	2355	Data 0880	2355	
SR	0000	SR	0008	(N=1)

Example 2

```
NEG  0x1200         ; Negate (0x1200) (Word mode)
```

	Before Instruction		After Instruction
Data 1200	8923	Data 1200	76DD
SR	0000	SR	0000

NEG

Negate Ws

Syntax:	{label:}	NEG{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: $\overline{(Ws)} + 1 \rightarrow Wd$

Status Affected: DC, N, OV, Z, C

Encoding:

1110	1010	0Bqq	qddd	dppp	ssss

Description: Compute the 2's complement of the contents of the source register Ws and place the result in the destination register Wd. Either register direct or indirect addressing may be used for both Ws and Wd.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1
```
NEG.B  W3, [W4++]   ; Negate W3 and store to [W4] (Byte mode)
                    ; Post-increment W4
```

	Before Instruction		After Instruction	
W3	7839	W3	7839	
W4	1005	W4	1006	
Data 1004	2355	Data 1004	C755	
SR	0000	SR	0008	(N=1)

Example 2
```
NEG   [W2++], [--W4]   ; Pre-decrement W4 (Word mode)
                       ; Negate [W2] and store to [W4]
                       ; Post-increment W2
```

	Before Instruction		After Instruction
W2	0900	W2	0902
W4	1002	W4	1000
Data 0900	870F	Data 0900	870F
Data 1000	5105	Data 1000	78F1
SR	0000	SR	0000

NEG

Negate Accumulator

Syntax: {label:} NEG Acc

Operands: Acc ∈ [A,B]

Operation: If (Acc = A):
-ACCA → ACCA
Else:
-ACCB → ACCB

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1100	1011	A001	0000	0000	0000

Description: Compute the 2's complement of the contents of the specified accumulator. Regardless of the Saturation mode, this instruction operates on all 40-bits of the accumulator.

The 'A' bit specifies the selected accumulator.

Words: 1

Cycles: 1

Example 1

```
NEG  A     ; Negate ACCA
           ; Store result to ACCA
           ; CORCON = 0x0000 (no saturation)
```

	Before Instruction	After Instruction
ACCA	00 3290 59C8	FF CD6F A638
CORCON	0000	0000
SR	0000	0000

Example 2

```
NEG  B     ; Negate ACCB
           ; Store result to ACCB
           ; CORCON = 0x00C0 (normal saturation)
```

	Before Instruction	After Instruction
ACCB	FF F230 10DC	00 0DCF EF24
CORCON	00C0	00C0
SR	0000	0000

NOP

No Operation

Syntax:	{label:}	NOP
Operands:	None	
Operation:	No Operation	
Status Affected:	None	

Encoding:

0000	0000	xxxx	xxxx	xxxx	xxxx

Description:	No Operation is performed. The 'x' bits can take any value.
Words:	1
Cycles:	1

Example 1

```
NOP      ; execute no operation
```

	Before Instruction	After Instruction
PC	00 1092	00 1094
SR	0000	0000

Example 2

```
NOP      ; execute no operation
```

	Before Instruction	After Instruction
PC	00 08AE	00 08B0
SR	0000	0000

NOPR — No Operation

Syntax:	{label:}	NOPR
Operands:	None	
Operation:	No Operation	
Status Affected:	None	
Encoding:	1111 1111 xxxx xxxx xxxx xxxx	
Description:	No Operation is performed. The ‘x’ bits can take any value.	
Words:	1	
Cycles:	1	

Example 1

```
NOPR    ; execute no operation
```

	Before Instruction	After Instruction
PC	00 2430	00 2432
SR	0000	0000

Example 2

```
NOPR    ; execute no operation
```

	Before Instruction	After Instruction
PC	00 1466	00 1468
SR	0000	0000

POP

Pop TOS to f

Syntax:	{label:} POP f
Operands:	f ∈ [0 ... 65534]
Operation:	(W15)-2 → W15 (TOS) → f
Status Affected:	None
Encoding:	`1111` `1001` `ffff` `ffff` `ffff` `fff0`
Description:	The stack pointer (W15) is pre-decremented by 2 and the Top-of-Stack (TOS) word is written to the specified file register, which may reside anywhere in the lower 32K words of data memory. The 'f' bits select the address of the file register. **Note 1:** This instruction operates in Word mode only. **2:** The file register address must be word aligned.
Words:	1
Cycles:	1

Example 1

```
POP  0x1230    ; Pop TOS to 0x1230
```

	Before Instruction		After Instruction
W15	1006	W15	1004
Data 1004	A401	Data 1004	A401
Data 1230	2355	Data 1230	A401
SR	0000	SR	0000

Example 2

```
POP  0x880     ; Pop TOS to 0x880
```

	Before Instruction		After Instruction
W15	2000	W15	1FFE
Data 0880	E3E1	Data 0880	A090
Data 1FFE	A090	Data 1FFE	A090
SR	0000	SR	0000

POP

Pop TOS to Wd

Syntax:	{label:}	POP	Wd
			[Wd]
			[Wd++]
			[Wd--]
			[--Wd]
			[++Wd]
			[Wd+Wb]

Operands: Wd ∈ [W0 ... W15]
Wb ∈ [W0 ... W15]

Operation: (W15)-2 → W15
(TOS) → Wd

Status Affected: None

Encoding:

0111	1www	w0hh	hddd	d100	1111

Description: The stack pointer (W15) is pre-decremented by 2 and the Top-of-Stack (TOS) word is written to Wd. Either register direct or indirect addressing may be used for Wd.

The ‘w’ bits define the offset register Wb.
The ‘h’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.

Note 1: This instruction operates in Word mode only.
2: This instruction is a specific version of the “`MOV Ws, Wd`” instruction (`MOV [--W15], Wd`). It reverse assembles as `MOV`.

Words: 1

Cycles: 1

Example 1

```
POP  W4        ; Pop TOS to W4
```

	Before Instruction		After Instruction
W4	EDA8	W4	C45A
W15	1008	W15	1006
Data 1006	C45A	Data 1006	C45A
SR	0000	SR	0000

Example 2

```
POP  [++W10]   ; Pre-increment W10
               ; Pop TOS to [W10]
```

	Before Instruction		After Instruction
W10	0E02	W10	0E04
W15	1766	W15	1764
Data 0E04	E3E1	Data 0E04	C7B5
Data 1764	C7B5	Data 1764	C7B5
SR	0000	SR	0000

POP.D

Double Pop TOS to Wnd:Wnd+1

Syntax:	{label:} POP.D Wnd
Operands:	Wnd ∈ [W0, W2, W4, ... W14]
Operation:	(W15)-2 → W15 (TOS) → Wnd+1 (W15)-2 → W15 (TOS) → Wnd
Status Affected:	None
Encoding:	1011 1110 0000 0ddd 0100 1111
Description:	A double-word is popped from the Top-of-Stack (TOS) and stored to Wnd:Wnd+1. The Most Significant Word is stored to Wnd+1, and the Least Significant Word is stored to Wnd. Since a double-word is popped, the stack pointer (W15) gets decremented by 4. The 'd' bits select the address of the destination register pair. **Note 1:** This instruction operates on double-words. See Figure 4-2 for information on how double-words are aligned in memory. **2:** Wnd must be an even working register. **3:** This instruction is a specific version of the "`MOV.D Ws, Wnd`" instruction (`MOV.D [--W15], Wnd`). It reverse assembles as `MOV.D`.
Words:	1
Cycles:	2

Example 1

```
POP.D  W6        ; Double pop TOS to W6
```

	Before Instruction	After Instruction
W6	07BB	3210
W7	89AE	7654
W15	0850	084C
Data 084C	3210	3210
Data 084E	7654	7654
SR	0000	0000

Example 2

```
POP.D  W0        ; Double pop TOS to W0
```

	Before Instruction	After Instruction
W0	673E	791C
W1	DD23	D400
W15	0BBC	0BB8
Data 0BB8	791C	791C
Data 0BBA	D400	D400
SR	0000	0000

POP.S

Pop Shadow Registers

Syntax:	{label:}	POP.S

Operands: None

Operation: Pop shadow registers

Status Affected: DC, N, OV, Z, C

Encoding:

1111	1110	1000	0000	0000	0000

Description: The values in the shadow registers are copied into their respective primary registers. The following registers are affected: W0-W3, and the C, Z, OV, N and DC Status register flags.

Note 1: The shadow registers are not directly accessible. They may only be accessed with `PUSH.S` and `POP.S`.

2: The shadow registers are only one-level deep.

Words: 1

Cycles: 1

Example 1

```
POP.S      ; Pop the shadow registers
           ; (See PUSH.S Example 1 for contents of shadows)
```

	Before Instruction		After Instruction
W0	07BB	W0	0000
W1	03FD	W1	1000
W2	9610	W2	2000
W3	7249	W3	3000
SR	00E0 (IPL=7)	SR	00E1 (IPL=7, C=1)

Note: After instruction execution, contents of shadow registers are NOT modified.

5

Instruction Descriptions

PUSH — Push f to TOS

Syntax:	{label:} PUSH f
Operands:	f ∈ [0 ... 65534]
Operation:	(f) → (TOS) (W15)+2 → W15
Status Affected:	None
Encoding:	`1111` `1000` `ffff` `ffff` `ffff` `fff0`
Description:	The contents of the specified file register are written to the Top-of-Stack (TOS) location and then the stack pointer (W15) is incremented by 2. The file register may reside anywhere in the lower 32K words of data memory. The 'f' bits select the address of the file register. **Note 1:** This instruction operates in Word mode only. **2:** The file register address must be word aligned.
Words:	1
Cycles:	1

Example 1

```
PUSH   0x2004        ; Push (0x2004) to TOS
```

	Before Instruction		After Instruction
W15	0B00	W15	0B02
Data 0B00	791C	Data 0B00	D400
Data 2004	D400	Data 2004	D400
SR	0000	SR	0000

Example 2

```
PUSH   0xC0E        ; Push (0xC0E) to TOS
```

	Before Instruction		After Instruction
W15	0920	W15	0922
Data 0920	0000	Data 0920	67AA
Data 0C0E	67AA	Data 2004	67AA
SR	0000	SR	0000

PUSH

Push Ws to TOS

Syntax:	{label:}	PUSH	Ws
			[Ws]
			[Ws++]
			[Ws--]
			[--Ws]
			[++Ws]
			[Ws+Wb]

Operands: Ws ∈ [W0 ... W15]
Wb ∈ [W0 ... W15]

Operation: (Ws) → (TOS)
(W15)+2 → W15

Status Affected: None

Encoding:

0111	1www	w001	1111	1ggg	ssss

Description: The contents of Ws are written to the Top-of-Stack (TOS) location and then the stack pointer (W15) is incremented by 2.

The 'w' bits define the offset register Wb.
The 'g' bits select the source Address mode.
The 's' bits select the address of the source register.

Note 1: This instruction operates in Word mode only.
2: This instruction is a specific version of the "`MOV Ws, Wd`" instruction (`MOV Ws, [W15++]`). It reverse assembles as `MOV`.

Words: 1

Cycles: 1

Example 1

```
PUSH  W2          ; Push W2 to TOS
```

	Before Instruction		After Instruction
W2	6889	W2	6889
W15	1566	W15	1568
Data 1566	0000	Data 1566	6889
SR	0000	SR	0000

Example 2

```
PUSH  [W5+W10]    ; Push [W5+W10] to TOS
```

	Before Instruction		After Instruction
W5	1200	W5	1200
W10	0044	W10	0044
W15	0806	W15	0808
Data 0806	216F	Data 0806	B20A
Data 1244	B20A	Data 1244	B20A
SR	0000	SR	0000

PUSH.D

Double Push Wns:Wns+1 to TOS

Syntax:	{label:} PUSH.D Wns
Operands:	Wns ∈ [W0, W2, W4 ... W14]
Operation:	(Wns) → (TOS) (W15)+2 → W15 (Wns+1) → (TOS) (W15)+2 → W15
Status Affected:	None

Encoding:

1011	1110	1001	1111	1000	sss0

Description: A double-word (Wns:Wns+1) is pushed to the Top-of-Stack (TOS). The Least Significant word (Wns) is pushed to the TOS first, and the Most Significant word (Wns+1) is pushed to the TOS last. Since a double-word is pushed, the stack pointer (W15) gets incremented by 4.

The 's' bits select the address of the source register pair.

Note 1: This instruction operates on double-words. See Figure 4-2 for information on how double-words are aligned in memory.
2: Wns must be an even working register.
3: This instruction is a specific version of the "`MOV.D Wns, Wd`" instruction (`MOV.D Wns, [W15++]`). It reverse assembles as `MOV.D`.

Words: 1

Cycles: 2

Example 1 `PUSH.D W6 ; Push W6:W7 to TOS`

	Before Instruction		After Instruction
W6	C451	W6	C451
W7	3380	W7	3380
W15	1240	W15	1244
Data 1240	B004	Data 1240	C451
Data 1242	0891	Data 1242	3380
SR	0000	SR	0000

Example 2 `PUSH.D W10 ; Push W10:W11 to TOS`

	Before Instruction		After Instruction
W10	80D3	W10	80D3
W11	4550	W11	4550
W15	0C08	W15	0C0C
Data 0C08	79B5	Data 0C08	80D3
Data 0C0A	008E	Data 0C0A	4550
SR	0000	SR	0000

PUSH.S — Push Shadow Registers

Syntax:	{label:}	PUSH.S
Operands:	None	
Operation:	Push shadow registers	
Status Affected:	None	

Encoding:

1111	1110	1010	0000	0000	0000

Description: The contents of the primary registers are copied into their respective shadow registers. The following registers are shadowed: W0-W3, and the C, Z, OV, N and DC Status register flags.

Note 1: The shadow registers are not directly accessible. They may only be accessed with `PUSH.S` and `POP.S`.

2: The shadow registers are only one-level deep.

Words: 1

Cycles: 1

Example 1
```
PUSH.S    ; Push primary registers into shadow registers
```

	Before Instruction		After Instruction	
W0	0000		0000	
W1	1000		1000	
W2	2000		2000	
W3	3000		3000	
SR	0001	(C=1)	0001	(C=1)

Note: After an instruction execution, contents of the shadow registers are updated.

PWRSAV — Enter Power Saving Mode

Syntax:	{label:} PWRSAV #lit1
Operands:	lit1 ∈ [0,1]
Operation:	0 → WDT count register 0 → WDT prescaler A count 0 → WDT prescaler B count 0 → WDTO (RCON<4>) 0 → SLEEP (RCON<3>) 0 → IDLE (RCON<2>) If (lit1 = 0): Enter SLEEP mode Else: Enter IDLE mode
Status Affected:	None
Encoding:	1111 1110 0100 0000 0000 000k
Description:	Place the processor into the specified Power Saving mode. If lit1 = 0, SLEEP mode is entered. In SLEEP mode, the clock to the CPU and peripherals are shutdown. If an on-chip oscillator is being used, it is also shutdown. If lit1 = 1, IDLE mode is entered. In IDLE mode, the clock to the CPU shuts down, but the clock source remains active and the peripherals continue to operate. This instruction resets the Watchdog Timer Count register and the Prescaler Count registers. In addition, the WDTO, SLEEP and IDLE flags of the Reset System and Control (RCON) register are reset. **Note 1:** The processor will exit from IDLE or SLEEP through an interrupt, processor RESET or Watchdog Time-out. See the dsPIC30F Data Sheet for details. **2:** If awakened from IDLE mode, IDLE (RCON<2>) is set to ‘1’ and the clock source is applied to the CPU. **3:** If awakened from SLEEP mode, SLEEP (RCON<3>) is set to ‘1’ and the clock source is started. **4:** If awakened from a Watchdog Time-out, WDTO (RCON<4>) is set to ‘1’.
Words:	1
Cycles:	1

Example 1

```
PWRSAV  #0      ; Enter SLEEP mode
```

	Before Instruction	After Instruction
SR	0040 (IPL=2)	0040 (IPL=2)

Example 2

```
PWRSAV  #1      ; Enter IDLE mode
```

	Before Instruction	After Instruction
SR	0020 (IPL=1)	0020 (IPL=1)

RCALL

Relative Call

Syntax:	{label:}	RCALL	Expr	

Operands: Expr may be an absolute address, label or expression.
Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... 32767].

Operation:
(PC) + 2 → PC
(PC<15:0>) → (TOS)
(W15) + 2 → W15
(PC<22:16>) → (TOS)
(W15) + 2 → W15
(PC) + (2 * Slit16) → PC
NOP → Instruction Register

Status Affected: None

Encoding:

0000	0111	nnnn	nnnn	nnnn	nnnn

Description: Relative subroutine call with a range of 32K program words forward or back from the current PC. Before the call is made, the return address (PC+2) is pushed onto the stack. After the return address is stacked, the sign-extended 17-bit value (2 * Slit16) is added to the contents of the PC and the result is stored in the PC.

The 'n' bits are a signed literal that specifies the size of the relative call (in program words) from (PC+2).

Note: When possible, this instruction should be used instead of `CALL`, since it only consumes one word of program memory.

Words: 1

Cycles: 2

Example 1

```
012004          RCALL  _Task1         ; Call _Task1
012006          ADD    W0, W1, W2
   .            ...
   .            ...
012458 _Task1:  SUB    W0, W2, W3     ; _Task1 subroutine
01245A          ...
```

	Before Instruction		After Instruction
PC	01 2004	PC	01 2458
W15	0810	W15	0814
Data 0810	FFFF	Data 0810	2006
Data 0812	FFFF	Data 0812	0001
SR	0000	SR	0000

Example 2

```
00620E          RCALL   _Init         ; Call _Init
006210          MOV     W0, [W4++]
   .            ...
   .            ...
007000 _Init:   CLR     W2            ; _Init subroutine
007002          ...
```

	Before Instruction		After Instruction
PC	00 620E	PC	00 7000
W15	0C50	W15	0C54
Data 0C50	FFFF	Data 0C50	6210
Data 0C52	FFFF	Data 0C52	0000
SR	0000	SR	0000

RCALL — Computed Relative Call

Syntax: {label:} RCALL Wn

Operands: Wn ∈ [W0 ... W15]

Operation: (PC) + 2 → PC
(PC<15:0>) → (TOS)
(W15) + 2 → W15
(PC<22:16>) → (TOS)
(W15) + 2 → W15
(PC) + (2 * (Wn)) → PC
NOP → Instruction Register

Status Affected: None

Encoding:

0000	0001	0010	0000	0000	ssss

Description: Computed, relative subroutine call specified by the working register Wn. The range of the call is 32K program words forward or back from the current PC. Before the call is made, the return address (PC+2) is pushed onto the stack. After the return address is stacked, the sign-extended 17-bit value (2 * (Wn)) is added to the contents of the PC and the result is stored in the PC. Register direct addressing must be used for Wn.

The ‘s’ bits select the address of the source register.

Words: 1

Cycles: 2

Example 1

```
00FF8C  EX1:    INC     W2, W3          ; Destination of RCALL
00FF8E          ...
   .            ...
   .            ...
010008
01000A          RCALL   W6              ; RCALL with W6
01000C          MOVE    W4, [W10]
```

	Before Instruction		After Instruction
PC	01 000A	PC	00 FF8C
W6	FFC0	W6	FFC0
W15	1004	W15	1008
Data 1004	98FF	Data 1004	000C
Data 1006	2310	Data 1006	0001
SR	0000	SR	0000

Example 2

```
000302          RCALL    W2             ; RCALL with W2
000304          FF1L     W0, W1
   .            ...
   .            ...
000450  EX2:    CLR      W2             ; Destination of RCALL
000452          ...
```

	Before Instruction		After Instruction
PC	00 0302	PC	00 0450
W2	00A6	W2	00A6
W15	1004	W15	1008
Data 1004	32BB	Data 1004	0304
Data 1006	901A	Data 1006	0000
SR	0000	SR	0000

REPEAT

Repeat Next Instruction 'lit14+1' Times

Syntax: {label:} REPEAT #lit14

Operands: lit14 ∈ [0 ... 16383]

Operation: (lit14) → RCOUNT
(PC)+2 → PC
Enable Code Looping

Status Affected: RA

Encoding:

0000	1001	00kk	kkkk	kkkk	kkkk

Description: Repeat the instruction immediately following the `REPEAT` instruction (lit14 + `1`) times. The repeated instruction (or target instruction) is held in the instruction register for all iterations and is only fetched once.

When this instruction executes, the RCOUNT register is loaded with the repeat count value specified in the instruction. RCOUNT is decremented with each execution of the target instruction. When RCOUNT equals zero, the target instruction is executed one more time, and then normal instruction execution continues with the instruction following the target instruction.

The 'k' bits are an unsigned literal that specifies the loop count.

Special Features, Restrictions:

1. When the repeat literal is '`0`', `REPEAT` has the effect of a `NOP` and the RA bit is not set.
2. The target `REPEAT` instruction can NOT be:
   - an instruction that changes program flow
   - a `DO, DISI, LNK, MOV.D, PWRSAV, REPEAT` or `UNLK` instruction
   - a 2-word instruction

   Unexpected results may occur if these target instructions are used.

Note: The `REPEAT` and target instruction are interruptible.

Words: 1

Cycles: 1

Example 1

```
000452   REPEAT #9                  ; Execute ADD 10 times
000454   ADD    [W0++], W1, [W2++]  ; Vector update
```

	Before Instruction		After Instruction	
PC	00 0452	PC	00 0454	
RCOUNT	0000	RCOUNT	0009	
SR	0000	SR	0010	(RA=1)

Example 2

```
00089E   REPEAT  #0x3FF    ; Execute CLR 1024 times
0008A0   CLR     [W6++]    ; Clear the scratch space
```

	Before Instruction		After Instruction	
PC	00 089E	PC	00 08A0	
RCOUNT	0000	RCOUNT	03FF	
SR	0000	SR	0010	(RA=1)

REPEAT

Repeat Next Instruction Wn+1 Times

Syntax:	{label:} REPEAT Wn
Operands:	Wn ∈ [W0 ... W15]
Operation:	(Wn<13:0>) → RCOUNT (PC)+2 → PC Enable Code Looping
Status Affected:	RA
Encoding:	0000 1001 1000 0000 0000 ssss

Description: Repeat the instruction immediately following the `REPEAT` instruction (Wn<13:0>) times. The instruction to be repeated (or target instruction) is held in the instruction register for all iterations and is only fetched once.

When this instruction executes, the RCOUNT register is loaded with the lower 14-bits of Wn. RCOUNT is decremented with each execution of the target instruction. When RCOUNT equals zero, the target instruction is executed one more time, and then normal instruction execution continues with the instruction following the target instruction.

The 's' bits specify the Wn register that contains the repeat count.

Special Features, Restrictions:

1. When (Wn) = `0`, `REPEAT` has the effect of a `NOP` and the RA bit is not set.
2. The target `REPEAT` instruction can NOT be:
   - an instruction that changes program flow
   - a `DO, DISI, LNK, MOV.D, PWRSAV, REPEAT` or `ULNK` instruction
   - a 2-word instruction

   Unexpected results may occur if these target instructions are used.

Note: The `REPEAT` and target instruction are interruptible.

Words: 1

Cycles: 1

Example 1

```
000A26  REPEAT  W4              ; Execute COM (W4+1) times
000A28  COM     [W0++], [W2++]  ; Vector complement
```

	Before Instruction		After Instruction	
PC	00 0A26	PC	00 0A28	
W4	0023	W4	0023	
RCOUNT	0000	RCOUNT	0023	
SR	0000	SR	0010	(RA=1)

Example 2

```
00089E  REPEAT  W10             ; Execute TBLRD (W10+1) times
0008A0  TBLRDL  [W2++], [W3++]  ; Decrement (0x840)
```

	Before Instruction		After Instruction	
PC	00 089E	PC	00 08A0	
W10	00FF	W10	00FF	
RCOUNT	0000	RCOUNT	00FF	
SR	0000	SR	0010	(RA=1)

RESET

Reset

Syntax:	{label:} RESET
Operands:	None
Operation:	Force all registers that are affected by a $\overline{\text{MCLR}}$ Reset to their RESET condition. 1 → SWR (RCON<6>) 0 → PC
Status Affected:	OA, OB, OAB, SA, SB, SAB, DA, DC, IPL<2:0>, RA, N, OV, Z, C
Encoding:	1111 1110 0000 0000 0000 0000
Description:	This instruction provides a way to execute a software RESET. All core and peripheral registers will take their power-on value. The PC will be set to '0', the location of the RESET GOTO instruction. The SWR bit, RCON<6>, will be set to '1' to indicate that the RESET instruction was executed.

Note: Refer to the dsPIC30F MCU Family Reference Manual for the power-on value of all registers.

Words:	1
Cycles:	1

Example 1

```
00202A   RESET      ; Execute software RESET
```

	Before Instruction			After Instruction	
PC	00 202A		PC	00 0000	
W0	8901		W0	0000	
W1	08BB		W1	0000	
W2	B87A		W2	0000	
W3	872F		W3	0000	
W4	C98A		W4	0000	
W5	AAD4		W5	0000	
W6	981E		W6	0000	
W7	1809		W7	0000	
W8	C341		W8	0000	
W9	90F4		W9	0000	
W10	F409		W10	0000	
W11	1700		W11	0000	
W12	1008		W12	0000	
W13	6556		W13	0000	
W14	231D		W14	0000	
W15	1704		W15	0800	
SPLIM	1800		SPLIM	0000	
TBLPAG	007F		TBLPAG	0000	
PSVPAG	0001		PSVPAG	0000	
CORCON	00F0		CORCON	0020	(SATDW=1)
RCON	0000		RCON	0040	(SWR=1)
SR	0021	(IPL, C=1)	SR	0000	

 Advance Information

RETFIE — Return from Interrupt

Syntax:	{label:}	RETFIE

Operands: None

Operation:
(W15)-2 $\rightarrow$ W15
(TOS<15:8>) $\rightarrow$ (SR<7:0>)
(TOS<7>) $\rightarrow$ (IPL3, CORCON<3>)
(TOS<6:0>) $\rightarrow$ (PC<22:16>)
(W15)-2 $\rightarrow$ W15
(TOS<15:0>) $\rightarrow$ (PC<15:0>)
NOP $\rightarrow$ Instruction Register

Status Affected: IPL<3:0>, RA, N, OV, Z, C

Encoding:

0000	0110	0100	0000	0000	0000

Description: Return from Interrupt Service Routine. The stack is popped, which loads the low byte of the Status register, IPL<3> (CORCON<3>) and the Most Significant Byte of the PC. The stack is popped again, which loads the lower 16 bits of the PC.

Note 1: Restoring IPL<3> and the low byte of the Status register restores the Interrupt Priority Level to the level before the execution was processed.

2: Before `RETFIE` is executed, the appropriate interrupt flag must be cleared in software to avoid recursive interrupts.

Words: 1

Cycles: 3 (2 if exception pending)

Example 1 `000A26   RETFIE   ; Return from ISR`

	Before Instruction		After Instruction	
PC	00 0A26	PC	01 0230	
W15	0834	W15	0830	
Data 0830	0230	Data 0830	0230	
Data 0832	8101	Data 0832	8101	
CORCON	0001	CORCON	0001	
SR	0000	SR	0081	(IPL=4, C=1)

Example 2 `008050   RETFIE   ; Return from ISR`

	Before Instruction		After Instruction	
PC	00 8050	PC	00 7008	
W15	0926	W15	0922	
Data 0922	7008	Data 0922	7008	
Data 0924	0300	Data 0924	0300	
CORCON	0000	CORCON	0000	
SR	0000	SR	0003	(Z, C=1)

RETLW

Return with Literal in Wn

Syntax:	{label:}	RETLW{.B} #lit10,	Wn

Operands: lit10 ∈ [0 ... 255] for byte operation
lit10 ∈ [0 ... 1023] for word operation
Wn ∈ [W0 ... W15]

Operation: (W15)-2 → W15
(TOS) → (PC<22:16>)
(W15)-2 → W15
(TOS) → (PC<15:0>)
lit10 → Wn

Status Affected: None

Encoding:

0000	0101	0Bkk	kkkk	kkkk	dddd

Description: Return from subroutine with the specified, unsigned 10-bit literal stored in Wn. The software stack is popped twice to restore the PC and the signed literal is stored in Wn. Since two pops are made, the stack pointer (W15) is decremented by 4.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'k' bits specify the value of the literal.
The 'd' bits select the address of the destination register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See **Section 4.6 "Using 10-bit Literal Operands"** for information on using 10-bit literal operands in Byte mode.

Words: 1

Cycles: 3 (2 if exception pending)

Example 1

```
000440   RETLW.B #0xA, W0   ; Return with 0xA in W0
```

	Before Instruction		After Instruction
PC	00 0440	PC	00 7006
W0	9846	W0	980A
W15	1988	W15	1984
Data 1984	7006	Data 1984	7006
Data 1986	0000	Data 1986	0000
SR	0000	SR	0000

Example 2

```
00050A   RETLW  #0x230, W2  ; Return with 0x230 in W2
```

	Before Instruction		After Instruction
PC	00 050A	PC	01 7008
W2	0993	W2	0230
W15	1200	W15	11FC
Data 11FC	7008	Data 11FC	7008
Data 11FE	0001	Data 11FE	0001
SR	0000	SR	0000

 Advance Information

RETURN — Return

Syntax:	{label:}	RETURN
Operands:	None	
Operation:	(W15)-2 → W15 (TOS) → (PC<22:16>) (W15)-2 → W15 (TOS) → (PC<15:0>) NOP → Instruction Register	
Status Affected:	None	

Encoding:

0000	0110	0000	0000	0000	0000

Description:	Return from subroutine. The software stack is popped twice to restore the PC. Since two pops are made, the stack pointer (W15) is decremented by 4.
Words:	1
Cycles:	3 (2 if exception pending)

Example 1

```
001A06    RETURN      ; Return from subroutine
```

	Before Instruction	After Instruction
PC	00 1A06	01 0004
W15	1248	1244
Data 1244	0004	0004
Data 1246	0001	0001
SR	0000	0000

Example 2

```
005404    RETURN      ; Return from subroutine
```

	Before Instruction	After Instruction
PC	00 5404	00 0966
W15	090A	0906
Data 0906	0966	0966
Data 0908	0000	0000
SR	0000	0000

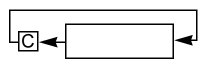
RLC

Rotate Left f through Carry

Syntax:	{label:}	RLC{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation:
For byte operation:
(C) → Dest<0>
(f<6:0>) → Dest<7:1>
(f<7>) → C
For word operation:
(C) → Dest<0>
(f<14:0>) → Dest<15:1>
(f<15>) → C



Status Affected: N, Z, C

Encoding:

1101	0110	1BDf	ffff	ffff	ffff

Description: Rotate the contents of the file register f one bit to the left through the Carry flag and place the result in the destination register. The Carry flag of the Status Register is shifted into the Least Significant bit of the destination, and it is then overwritten with the Most Significant bit of Ws.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for f, `1` for WREG).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
RLC.B  0x1233      ; Rotate Left w/ C (0x1233) (Byte mode)
```

	Before Instruction		After Instruction	
Data 1232	E807	Data 1232	D007	
SR	0000	SR	0009	(N, C=1)

Example 2

```
RLC  0x820, WREG  ; Rotate Left w/ C (0x820) (Word mode)
                  ; Store result in WREG
```

	Before Instruction			After Instruction	
WREG (W0)	5601		WREG (W0)	42DD	
Data 0820	216E		Data 0820	216E	
SR	0001	(C=1)	SR	0000	(C=0)

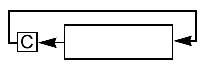
RLC

Rotate Left Ws through Carry

Syntax:	{label:}	RLC{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: For byte operation:
(C) → Wd<0>
(Ws<6:0>) → Wd<7:1>
(Ws<7>) → C
For word operation:
(C) → Wd<0>
(Ws<14:0>) → Wd<15:1>
(Ws<15>) → C



Status Affected: N, Z, C

Encoding:

1101	0010	1Bqq	qddd	dppp	ssss

Description: Rotate the contents of the source register Ws one bit to the left through the Carry flag and place the result in the destination register Wd. The Carry flag of the Status register is shifted into the Least Significant bit of Wd, and it is then overwritten with the Most Significant bit of Ws. Either register direct or indirect addressing may be used for Ws and Wd.

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1
```
RLC.B  W0, W3      ; Rotate Left w/ C (W0) (Byte mode)
                   ; Store the result in W3
```

	Before Instruction			After Instruction	
W0	9976		W0	9976	
W3	5879		W3	58ED	
SR	0001	(C=1)	SR	0009	(N=1)

5
Instruction Descriptions

Example 2

```
RLC   [W2++], [W8]   ; Rotate Left w/ C [W2] (Word mode)
                     ; Post-increment W2
                     ; Store result in [W8]
```

	Before Instruction			After Instruction	
W2	2008		W2	200A	
W8	094E		W8	094E	
Data 094E	3689		Data 094E	8082	
Data 2008	C041		Data 2008	C041	
SR	0001	(C=1)	SR	0009	(N, C=1)

 Advance Information

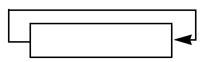
RLNC

Rotate Left f without Carry

Syntax:	{label:}	RLNC{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: <u>For byte operation:</u>
(f<6:0>) → Dest<7:1>
(f<7>) → Dest<0>
<u>For word operation:</u>
(f<14:0>) → Dest<15:1>
(f<15>) → Dest<0>



Status Affected: N, Z

Encoding:

1101	0110	0BDf	ffff	ffff	ffff

Description: Rotate the contents of the file register f one bit to the left and place the result in the destination register. The Most Significant bit of f is stored in the Least Significant bit of the destination, and the Carry flag is not affected.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
RLNC.B  0x1233     ; Rotate Left (0x1233) (Byte mode)
```

	Before Instruction			After Instruction	
Data 1232	E807		Data 1233	D107	
SR	0000		SR	0008	(N=1)

Example 2

```
RLNC  0x820, WREG  ; Rotate Left (0x820) (Word mode)
                   ; Store result in WREG
```

	Before Instruction			After Instruction	
WREG (W0)	5601		WREG (W0)	42DC	
Data 0820	216E		Data 0820	216E	
SR	0001	(C=1)	SR	0000	(C=0)

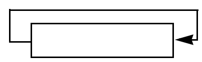
5
Instruction Descriptions

RLNC — Rotate Left Ws without Carry

Syntax:	{label:}	RLNC{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: For byte operation:
(Ws<6:0>) → Wd<7:1>
(Ws<7>) → Wd<0>
For word operation:
(Ws<14:0>) → Wd<15:1>
(Ws<15>) → Wd<0>



Status Affected: N, Z

Encoding:

1101	0010	0Bqq	qddd	dppp	ssss

Description: Rotate the contents of the source register Ws one bit to the left and place the result in the destination register Wd. The Most Significant bit of Ws is stored in the Least Significant bit of Wd, and the Carry flag is not affected. Either register direct or indirect addressing may be used for Ws and Wd.

The 'B' bit selects byte or word operation (`0` for byte, `1` for word).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
RLNC.B  W0, W3     ; Rotate Left (W0) (Byte mode)
                   ; Store the result in W3
```

	Before Instruction			After Instruction	
W0	9976		W0	9976	
W3	5879		W3	58EC	
SR	0001	(C=1)	SR	0009	(N, C=1)

Example 2

```
RLNC  [W2++], [W8] ; Rotate Left [W2] (Word mode)
                   ; Post-increment W2
                   ; Store result in [W8]
```

	Before Instruction			After Instruction	
W2	2008		W2	200A	
W8	094E		W8	094E	
Data 094E	3689		Data 094E	8083	
Data 2008	C041		Data 2008	C041	
SR	0001	(C=1)	SR	0009	(N, C=1)

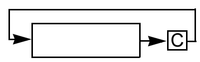
RRC

Rotate Right f through Carry

Syntax: {label:} RRC{.B} f {,WREG}

Operands: f ∈ [0 ... 8191]

Operation:
For byte operation:
(C) → Dest<7>
(f<7:1>) → Dest<6:0>
(f<0>) → C
For word operation:
(C) → Dest<15>
(f<15:1>) → Dest<14:0>
(f<0>) → C



Status Affected: N, Z, C

Encoding:

1101	0111	1BDf	ffff	ffff	ffff

Description: Rotate the contents of the file register f one bit to the right through the Carry flag and place the result in the destination register. The Carry flag of the Status Register is shifted into the Most Significant bit of the destination, and it is then overwritten with the Least Significant bit of Ws.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for byte, `1` for word).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
RRC.B  0x1233     ; Rotate Right w/ C (0x1233) (Byte mode)
```

	Before Instruction		After Instruction
Data 1232	E807	Data 1232	7407
SR	0000	SR	0000

Example 2

```
RRC  0x820, WREG  ; Rotate Right w/ C (0x820) (Word mode)
                  ; Store result in WREG
```

	Before Instruction			After Instruction	
WREG (W0)	5601		WREG (W0)	90B7	
Data 0820	216E		Data 0820	216E	
SR	0001	(C=1)	SR	0008	(N=1)

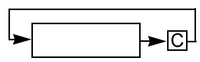
RRC

Rotate Right Ws through Carry

Syntax:	{label:}	RRC{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: For byte operation:
(C) → Wd<7>
(Ws<7:1>) → Wd<6:0>
(Ws<0>) → C
For word operation:
(C) → Wd<15>
(Ws<15:1>) → Wd<14:0>
(Ws<0>) → C



Status Affected: N, Z, C

Encoding:

1101	0011	1Bqq	qddd	dppp	ssss

Description: Rotate the contents of the source register Ws one bit to the right through the Carry flag and place the result in the destination register Wd. The Carry flag of the Status Register is shifted into the Most Significant bit of Wd, and it is then overwritten with the Least Significant bit of Ws. Either register direct or indirect addressing may be used for Ws and Wd.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

> **Note:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
RRC.B  W0, W3     ; Rotate Right w/ C (W0) (Byte mode)
                  ; Store the result in W3
```

	Before Instruction			After Instruction	
W0	9976		W0	9976	
W3	5879		W3	58BB	
SR	0001	(C=1)	SR	0008	(N=1)

Example 2

```
RRC  [W2++], [W8]  ; Rotate Right w/ C [W2] (Word mode)
                   ; Post-increment W2
                   ; Store result in [W8]
```

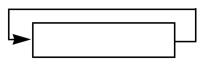
	Before Instruction			After Instruction	
W2	2008		W2	200A	
W8	094E		W8	094E	
Data 094E	3689		Data 094E	E020	
Data 2008	C041		Data 2008	C041	
SR	0001	(C=1)	SR	0009	(N, C=1)

 Advance Information

RRNC

Rotate Right f without Carry

Syntax:	{label:} RRNC{.B} f {,WREG}
Operands:	f ∈ [0 ... 8191]
Operation:	For byte operation: (f<7:1>) → Dest<6:0> (f<0>) → Dest<7> For word operation: (f<15:1>) → Dest<14:0> (f<0>) → Dest<15>



Status Affected: N, Z

Encoding:

1101	0111	0BDf	ffff	ffff	ffff

Description: Rotate the contents of the file register f one bit to the right and place the result in the destination register. The Least Significant bit of f is stored in the Most Significant bit of the destination, and the Carry flag is not affected.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
RRNC.B  0x1233     ; Rotate Right (0x1233) (Byte mode)
```

	Before Instruction		After Instruction
Data 1232	E807	Data 1232	7407
SR	0000	SR	0000

Example 2

```
RRNC   0x820, WREG  ; Rotate Right (0x820) (Word mode)
                    ; Store result in WREG
```

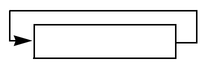
	Before Instruction			After Instruction	
WREG (W0)	5601		WREG (W0)	10B7	
Data 0820	216E		Data 0820	216E	
SR	0001	(C=1)	SR	0001	(C=1)

RRNC — Rotate Right Ws without Carry

Syntax:	{label:}	RRNC{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: <u>For byte operation:</u>
(Ws<7:1>) → Wd<6:0>
(Ws<0>) → Wd<7>
<u>For word operation:</u>
(Ws<15:1>) → Wd<14:0>
(Ws<0>) → Wd<15>



Status Affected: N, Z

Encoding:

1101	0011	0Bqq	qddd	dppp	ssss

Description: Rotate the contents of the source register Ws one bit to the right and place the result in the destination register Wd. The Least Significant bit of Ws is stored in the Most Significant bit of Wd, and the Carry flag is not affected. Either register direct or indirect addressing may be used for Ws and Wd.

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
RRNC.B  W0, W3      ; Rotate Right (W0) (Byte mode)
                    ; Store the result in W3
```

	Before Instruction			After Instruction	
W0	9976		W0	9976	
W3	5879		W3	583B	
SR	0001	(C=1)	SR	0001	(C=1)

 Advance Information

Example 2

```
RRNC  [W2++], [W8]  ; Rotate Right [W2] (Word mode)
                    ; Post-increment W2
                    ; Store result in [W8]
```

	Before Instruction		After Instruction	
W2	2008	W2	200A	
W8	094E	W8	094E	
Data 094E	3689	Data 094E	E020	
Data 2008	C041	Data 2008	C041	
SR	0000	SR	0008	(N=1)

SAC Store Accumulator

Syntax:	{label:}	SAC	Acc,	{#Slit4,}	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[--Wd]
					[++Wd]
					[Wd+Wb]

Operands: Acc ∈ [A,B]
Slit4 ∈ [-8 ... +7]
Wb, Wd ∈ [W0 ... W15]

Operation: $Shift_{Slit4}$(Acc) (optional)
(Acc[31:16]) → Wd

Status Affected: None

Encoding:

1100	1100	Awww	wrrr	rhhh	dddd

Description: Perform an optional, signed 4-bit shift of the specified accumulator, then store the shifted contents of AccH (Acc[31:16]) to Wd. The shift range is -8:7, where a negative operand indicates an arithmetic left shift and a positive operand indicates an arithmetic right shift. Either register direct or indirect addressing may be used for Wd.

The 'A' bit specifies the source accumulator.
The 'w' bits specify the offset register Wb.
The 'r' bits encode the optional accumulator pre-shift.
The 'h' bits select the destination Address mode.
The 'd' bits specify the destination register Wd.

Note 1: This instruction does not modify the contents of Acc.
2: This instruction stores the truncated contents of Acc. The instruction `SAC.R` may be used to store the rounded accumulator contents.
3: If Data Write saturation is enabled (SATDW, CORCON<5>, = `1`), the value stored to Wd is subject to saturation after the optional shift is performed.

Words: 1

Cycles: 1

Example 1

```
SAC  A, #4, W5
; Right shift ACCA by 4
; Store result to W5
; CORCON = 0x0010 (SATDW = 1)
```

	Before Instruction		After Instruction
W5	B900	W5	0120
ACCA	00 120F FF00	ACCA	00 120F FF00
CORCON	0010	CORCON	0010
SR	0000	SR	0000

Example 2

```
SAC  B, #-4, [W5++]
; Left shift ACCB by 4
; Store result to [W5], Post-increment W5
; CORCON = 0x0010 (SATDW = 1)
```

	Before Instruction		After Instruction
W5	2000	W5	2002
ACCB	FF C891 8F4C	ACCB	FF C891 1F4C
Data 2000	5BBE	Data 2000	8000
CORCON	0010	CORCON	0010
SR	0000	SR	0000

SAC.R — Store Rounded Accumulator

Syntax:	{label:}	SAC.R	Acc,	{#Slit4,}	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[--Wd]
					[++Wd]
					[Wd+Wb]

Operands: Acc ∈ [A,B]
Slit4 ∈ [-8 ... +7]
Wb ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: $Shift_{Slit4}$(Acc) (optional)
Round(Acc)
(Acc[31:16]) → Wd

Status Affected: None

Encoding:

1100	1101	Awww	wrrr	rhhh	dddd

Description: Perform an optional, signed 4-bit shift of the specified accumulator, then store the rounded contents of AccH (Acc[31:16]) to Wd. The shift range is -8:7, where a negative operand indicates an arithmetic left shift and a positive operand indicates an arithmetic right shift. The Rounding mode (Conventional or Convergent) is set by the RND bit, CORCON<1>. Either register direct or indirect addressing may be used for Wd.

The 'A' bit specifies the source accumulator.
The 'w' bits specify the offset register Wb.
The 'r' bits encode the optional accumulator pre-shift.
The 'h' bits select the destination Address mode.
The 'd' bits specify the destination register Wd.

Note 1: This instruction does not modify the contents of the Acc.
2: This instruction stores the rounded contents of Acc. The instruction `SAC` may be used to store the truncated accumulator contents.
3: If Data Write saturation is enabled (SATDW, CORCON<5>, = `1`), the value stored to Wd is subject to saturation after the optional shift is performed.

Words: 1

Cycles: 1

Example 1

```
SAC.R  A, #4, W5
; Right shift ACCA by 4
; Store rounded result to W5
; CORCON = 0x0010 (SATDW = 1)
```

	Before Instruction	After Instruction
W5	B900	0121
ACCA	00 120F FF00	00 120F FF00
CORCON	0010	0010
SR	0000	0000

 Advance Information

Example 2

```
SAC.R  B, #-4, [W5++]
; Left shift ACCB by 4
; Store rounded result to [W5], Post-increment W5
; CORCON = 0x0010 (SATDW = 1)
```

	Before Instruction		After Instruction
W5	2000	W5	2002
ACCB	FF F891 8F4C	ACCB	FF F891 8F4C
Data 2000	5BBE	Data 2000	8919
CORCON	0010	CORCON	0010
SR	0000	SR	0000

SE — Sign-Extend Ws

Syntax:	{label:}	SE	Ws,	Wnd
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
Wnd ∈ [W0 ... W15]

Operation: Ws<7:0> → Wnd<7:0>
If (Ws<7> = `1`):
  `0xFF` → Wnd<15:8>
Else:
  `0` → Wnd<15:8>

Status Affected: N, Z, C

Encoding:

1111	1011	0000	0ddd	dppp	ssss

Description: Sign-extend the byte in Ws and store the 16-bit result in Wnd. Either register direct or indirect addressing may be used for Ws, and register direct addressing must be used for Wnd. The C flag is set to the complement of the N flag.

The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note 1: This operation converts a byte to a word, and it uses no `.B` or `.W` extension.

2: The source Ws is addressed as a byte operand, so any address modification is by '`1`'.

Words: 1

Cycles: 1

Example 1
```
SE   W3, W4    ; Sign-extend W3 and store to W4
```

	Before Instruction	After Instruction	
W3	7839	7839	
W4	1005	0039	
SR	0000	0001	(C=1)

Example 2
```
SE   [W2++], W12  ; Sign-extend [W2] and store to W12
                  ; Post-increment W2
```

	Before Instruction	After Instruction	
W2	0900	0901	
W12	1002	FF8F	
Data 0900	008F	008F	
SR	0000	0008	(N=1)

SETM — Set f or WREG

Syntax:	{label:}	SETM{.B}	f
			WREG

Operands: f ∈ [0 ... 8191]

Operation:
For byte operation:
`0xFF` → destination designated by D
For word operation:
`0xFFFF` → destination designated by D

Status Affected: None

Encoding:

1110	1111	1BDf	ffff	ffff	ffff

Description: All the bits of the specified register are set to '1'. If WREG is specified, the bits of WREG are set. Otherwise, the bits of the specified file register are set.

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'D' bit selects the destination (0 for WREG, 1 for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1 `SETM.B 0x891 ; Set 0x891 (Byte mode)`

	Before Instruction		After Instruction
Data 0890	2739	Data 0890	FF39
SR	0000	SR	0000

Example 2 `SETM WREG ; Set WREG (Word mode)`

	Before Instruction		After Instruction
WREG (W0)	0900	WREG (W0)	FFFF
SR	0000	SR	0000

SETM — Set Ws

Syntax:	{label:}	SETM{.B}	Wd
			[Wd]
			[Wd++]
			[Wd--]
			[++Wd]
			[--Wd]

Operands: Wd ∈ [W0 ... W15]

Operation: For byte operation:
`0xFF` → Wd for byte operation
For word operation:
`0xFFFF` → Wd for word operation

Status Affected: None

Encoding:

1110	1011	1Bqq	qddd	d000	0000

Description: All the bits of the specified register are set to '1'. Either register direct or indirect addressing may be used for Wd.

The 'B' bits selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1
```
SETM.B   W13     ; Set W13 (Byte mode)
```

	Before Instruction		After Instruction
W13	2739	W13	27FF
SR	0000	SR	0000

Example 2
```
SETM     [--W6]  ; Pre-decrement W6 (Word mode)
                 ; Set [W6]
```

	Before Instruction		After Instruction
W6	1250	W6	124E
Data 124E	3CD9	Data 124E	FFFF
SR	0000	SR	0000

SFTAC

Arithmetic Shift Accumulator by Slit6

Syntax:	{label:}	SFTAC	Acc,	#Slit6

Operands: Acc ∈ [A,B]
Slit6 ∈ [-16 ... 16]

Operation: $Shift_k(Acc) \rightarrow Acc$

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1100	1000	A000	0000	01kk	kkkk

Description: Arithmetic shift the 40-bit contents of the specified accumulator by the signed, 6-bit literal and store the result back into the accumulator. The shift range is -16:16, where a negative operand indicates a left shift and a positive operand indicates a right shift. Any bits which are shifted out of the accumulator are lost.

The ‘A’ bit selects the accumulator for the result.
The ‘k’ bits determine the number of bits to be shifted.

Note 1: If saturation is enabled for the target accumulator (SATA, CORCON<6> or SATB, CORCON<7>), the value stored to the accumulator is subject to saturation.

2: If the shift amount is greater than 16 or less than -16, no modification will be made to the accumulator, and an arithmetic trap will occur.

Words: 1

Cycles: 1

Example 1

```
SFTAC  A, #12
; Arithmetic right shift ACCA by 12
; Store result to ACCA
; CORCON = 0x0080 (SATA = 1)
```

	Before Instruction		After Instruction
ACCA	00 120F FF00	ACCA	00 0001 20FF
CORCON	0080	CORCON	0080
SR	0000	SR	0000

Example 2

```
SFTAC  B, #-10
; Arithmetic left shift ACCB by 10
; Store result to ACCB
; CORCON = 0x0040 (SATB = 1)
```

	Before Instruction		After Instruction
ACCB	FF FFF1 8F4C	ACCB	FF C63D 3000
CORCON	0040	CORCON	0040
SR	0000	SR	0000

SFTAC — Arithmetic Shift Accumulator by Wb

Syntax:	{label:}	SFTAC	Acc,	Wb	

Operands: Acc ∈ [A,B]
Wb ∈ [W0 ... W15]

Operation: $\text{Shift}_{(Wb)}(\text{Acc}) \rightarrow \text{Acc}$

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1100	1000	A000	0000	0000	ssss

Description: Arithmetic shift the 40-bit contents of the specified accumulator and store the result back into the accumulator. The Least Significant 6 bits of Wb are used to specify the shift amount. The shift range is -16:16, where a negative value indicates a left shift and a positive value indicates a right shift. Any bits which are shifted out of the accumulator are lost.

The 'A' bit selects the accumulator for the source/destination.
The 's' bits select the address of the shift count register.

Note 1: If saturation is enabled for the target accumulator (SATA, CORCON<6> or SATB, CORCON<7>), the value stored to the accumulator is subject to saturation.

2: If the shift amount is greater than 16 or less than -16, no modification will be made to the accumulator, and an arithmetic trap will occur.

Words: 1

Cycles: 1

Example 1

```
SFTAC  A, W0
; Arithmetic shift ACCA by (W0)
; Store result to ACCA
; CORCON = 0x0000 (saturation disabled)
```

	Before Instruction		After Instruction	
W0	FFFC	W0	FFFC	
ACCA	00 320F AB09	ACCA	03 20FA B090	
CORCON	0000	CORCON	0000	
SR	0000	SR	8800	(OA, OAB=1)

Example 2

```
SFTAC  B, W12
; Arithmetic shift ACCB by (W12)
; Store result to ACCB
; CORCON = 0x0040 (SATB = 1)
```

	Before Instruction		After Instruction
W12	000F	W12	000F
ACCB	FF FFF1 8F4C	ACCB	FF FFFF FFE3
CORCON	0040	CORCON	0040
SR	0000	SR	0000

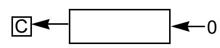
SL

Shift Left f

Syntax:	{label:}	SL{.B}	f	{,WREG}

Operands: f ∈ [0... 8191]

Operation:
For byte operation:
(f<7>) → (C)
(f<6:0>) → Dest<7:1>
0 → Dest<0>
For word operation:
(f<15>) → (C)
(f<14:0>) → Dest<15:1>
0 → Dest<0>



Status Affected: N, Z, C

Encoding:

1101	0100	0BDf	ffff	ffff	ffff

Description: Shift the contents of the file register one bit to the left and place the result in the destination register. The Most Significant bit of the file register is shifted into the Carry bit of the Status register, and zero is shifted into the Least Significant bit of the destination register.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘D’ bit selects the destination (`0` for WREG, `1` for file register).
The ‘f’ bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1
```
SL.B    0x909    ; Shift left (0x909) (Byte mode)
```

	Before Instruction		After Instruction	
Data 0908	9439	Data 0908	0839	
SR	0000	SR	0001	(C=1)

Example 2
```
SL      0x1650, WREG    ; Shift left (0x1650) (Word mode)
                        ; Store result in WREG
```

	Before Instruction		After Instruction	
WREG (W0)	0900	WREG (W0)	80CA	
Data 1650	4065	Data 1650	4065	
SR	0000	SR	0008	(N=1)

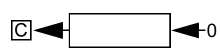
SL

Shift Left Ws

Syntax:	{label:}	SL{.B}	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation:
For byte operation:
(Ws<7>) → C
(Ws<6:0>) → Wd<7:1>
0 → Wd<0>
For word operation:
(Ws<15>) → C
(Ws<14:0>) → Wd<15:1>
0 → Wd<0>



Status Affected: N, Z, C

Encoding:

1101	0000	0Bqq	qddd	dppp	ssss

Description: Shift the contents of the source register Ws one bit to the left and place the result in the destination register Wd. The Most Significant bit of Ws is shifted into the Carry bit of the Status register, and 0 is shifted into the Least Significant bit of Wd. Either register direct or indirect addressing may be used for Ws and Wd.

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
SL.B  W3, W4    ; Shift left W3 (Byte mode)
                ; Store result to W4
```

	Before Instruction	After Instruction	
W3	78A9	78A9	
W4	1005	1052	
SR	0000	0001	(C=1)

Example 2

```
SL   [W2++], [W12]   ; Shift left [W2] (Word mode)
                     ; Store result to [W12]
                     ; Post-increment W2
```

	Before Instruction		After Instruction	
W2	0900	W2	0902	
W12	1002	W12	1002	
Data 0900	800F	Data 0900	800F	
Data 1002	6722	Data 1002	001E	
SR	0000	SR	0001	(C=1)

SL

Shift Left by Short Literal

Syntax:	{label:}	SL	Wb,	#lit4,	Wnd

Operands: Wb ∈ [W0 ... W15]
lit4 ∈ [0...15]
Wnd ∈ [W0 ... W15]

Operation: lit4<3:0> → Shift_Val
Wnd<15:Shift_Val> = Wb<15-Shift_Val:0>
Wd<Shift_Val-1:0> = 0

Status Affected: N, Z

Encoding:

1101	1101	0www	wddd	d100	kkkk

Description: Shift left the contents of the source register Wb by the 4-bit unsigned literal and store the result in the destination register Wnd. Any bits shifted out of the source register are lost. Direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the destination register.
The 'k' bits provide the literal operand, a five-bit integer number.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: 1

Example 1

```
SL    W2, #4, W2    ; Shift left W2 by 4
                    ; Store result to W2
```

	Before Instruction	After Instruction	
W2	78A9	8A90	
SR	0000	0008	(N=1)

Example 2

```
SL    W3, #12, W8   ; Shift left W3 by 12
                    ; Store result to W8
```

	Before Instruction	After Instruction
W3	0912	0912
W8	1002	2000
SR	0000	0000

SL

Shift Left by Wns

Syntax:	{label:}	SL	Wb,	Wns,	Wnd

Operands: Wb ∈ [W0 ... W15]
Wns ∈ [W0 ...W15]
Wnd ∈ [W0 ... W15]

Operation: Wns<4:0> → Shift_Val
Wnd<15:Shift_Val> = Wb<15-Shift_Val:0>
Wd<Shift_Val-1:0> = `0`

Status Affected: N, Z

Encoding:

1101	1101	0www	wddd	d000	ssss

Description: Shift left the contents of the source register Wb by the 5 Least Significant bits of Wns (only up to 15 positions) and store the result in the destination register Wnd. Any bits shifted out of the source register are lost. Register direct addressing must be used for Wb, Wns and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the destination register.
The 's' bits select the address of the source register.

Note 1: This instruction operates in Word mode only.
2: If Wns is greater than 15, Wnd will be loaded with `0x0`.

Words: 1

Cycles: 1

Example 1
```
SL    W0, W1, W2    ; Shift left W0 by W1<0:4>
                    ; Store result to W2
```

	Before Instruction	After Instruction
W0	09A4	09A4
W1	8903	8903
W2	78A9	4D20
SR	0000	0000

Example 2
```
SL    W4, W5, W6    ; Shift left W4 by W5<0:4>
                    ; Store result to W6
```

	Before Instruction	After Instruction
W4	A409	A409
W5	FF01	FF01
W6	0883	4812
SR	0000	0000

SUB

Subtract WREG from f

Syntax:	{label:}	SUB{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (f) – (WREG) → destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

1011	0101	0BDf	ffff	ffff	ffff

Description: Subtract the contents of the default working register WREG from the contents of the specified file register, and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
SUB.B 0x1FFF    ; Sub. WREG from (0x1FFF) (Byte mode)
                ; Store result to 0x1FFF
```

	Before Instruction	After Instruction	
WREG (W0)	7804	7804	
Data 1FFE	9439	9039	
SR	0000	0009	(N, C=1)

Example 2

```
SUB   0xA04, WREG   ; Sub. WREG from (0xA04) (Word mode)
                    ; Store result to WREG
```

	Before Instruction	After Instruction	
WREG (W0)	6234	E2EF	
Data 0A04	4523	4523	
SR	0000	0008	(N=1)

 Advance Information

SUB

Subtract Literal from Wn

Syntax:	{label:} SUB{.B} #lit10, Wn
Operands:	lit10 ∈ [0 ... 255] for byte operation lit10 ∈ [0 ... 1023] for word operation Wn ∈ [W0 ... W15]
Operation:	(Wn) – lit10 → Wn
Status Affected:	DC, N, OV, Z, C
Encoding:	`1011` `0001` `0Bkk` `kkkk` `kkkk` `dddd`
Description:	Subtract the 10-bit unsigned literal operand from the contents of the working register Wn, and store the result back in the working register Wn. Register direct addressing must be used for Wn.

The ‘B’ bit selects byte or word operation.
The ‘k’ bits specify the literal operand.
The ‘d’ bits select the address of the working register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See **Section 4.6 “Using 10-bit Literal Operands”** for information on using 10-bit literal operands in Byte mode.

Words: 1

Cycles: 1

Example 1

```
SUB.B   #0x23, W0    ; Sub. 0x23 from W0 (Byte mode)
                     ; Store result to W0
```

	Before Instruction	After Instruction	
W0	7804	78E1	
SR	0000	0008	(N=1)

Example 2

```
SUB     #0x108, W4   ; Sub. 0x108 from W4 (Word mode)
                     ; Store result to W4
```

	Before Instruction	After Instruction	
W4	6234	612C	
SR	0000	0001	(C=1)

SUB

Subtract Short Literal from Wb

Syntax:	{label:}	SUB{.B}	Wb,	#lit5,	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[++Wd]
					[--Wd]

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
Wd ∈ [W0 ... W15]

Operation: (Wb) – lit5 → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0101	0www	wBqq	qddd	d11k	kkkk

Description: Subtract the 5-bit unsigned literal operand from the contents of the base register Wb, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing must be used for Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'k' bits provide the literal operand, a five-bit integer number.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
SUB.B  W4, #0x10, W5   ; Sub. 0x10 from W4 (Byte mode)
                       ; Store result to W5
```

	Before Instruction		After Instruction
W4	1782	W4	1782
W5	7804	W5	7872
SR	0000	SR	0005 (OV, C=1)

Example 2

```
SUB    W0, #0x8, [W2++]   ; Sub. 0x8 from W0 (Word mode)
                          ; Store result to [W2]
                          ; Post-increment W2
```

	Before Instruction		After Instruction
W0	F230	W0	F230
W2	2004	W2	2006
Data 2004	A557	Data 2004	F228
SR	0000	SR	0009 (N, C=1)

SUB

Subtract Ws from Wb

Syntax:	{label:}	SUB{.B}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]
				[++Ws],	[++Wd]
				[--Ws],	[--Wd]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Wb) – (Ws) → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0101	0www	wBqq	qddd	dppp	ssss

Description: Subtract the contents of the source register Ws from the contents of the base register Wb and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
SUB.B  W0, W1, W0    ; Sub. W1 from W0 (Byte mode)
                     ; Store result to W0
```

	Before Instruction		After Instruction	
W0	1732	W0	17EE	
W1	7844	W1	7844	
SR	0000	SR	0108	(DC, N=1)

5

Instruction Descriptions

Example 2

```
SUB   W7, [W8++], [W9++]    ; Sub. [W8] from W7 (Word mode)
                            ; Store result to [W9]
                            ; Post-increment W8
                            ; Post-increment W9
```

	Before Instruction		After Instruction	
W7	2450	W7	2450	
W8	1808	W8	180A	
W9	2020	W9	2022	
Data 1808	92E4	Data 1808	92E4	
Data 2022	A557	Data 2022	916C	
SR	0000	SR	010C	(DC, N, OV=1)

SUB

Subtract Accumulators

Syntax:	{label:}	SUB	Acc		

Operands: Acc ∈ [A,B]

Operation: If (Acc = A):
ACCA – ACCB → ACCA
Else:
ACCB – ACCA → ACCB

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

1100	1011	A011	0000	0000	0000

Description: Subtract the contents of the unspecified accumulator from the contents of Acc, and store the result back into Acc. This instruction performs a 40-bit subtraction.

The ‘A’ bit specifies the destination accumulator.

Words: 1

Cycles: 1

Example 1

```
SUB  A      ; Subtract ACCB from ACCA
            ; Store the result to ACCA
            ; CORCON = 0x0000 (no saturation)
```

	Before Instruction		After Instruction	
ACCA	76 120F 098A	ACCA	52 1EFC 4D73	
ACCB	23 F312 BC17	ACCB	23 F312 BC17	
CORCON	0000	CORCON	0000	
SR	0000	SR	1100	(OA, OB=1)

Example 2

```
SUB  B      ; Subtract ACCA from ACCB
            ; Store the result to ACCB
            ; CORCON = 0x0040 (SATB = 1)
```

	Before Instruction		After Instruction	
ACCA	FF 9022 2EE1	ACCA	FF 9022 2EE1	
ACCB	00 2456 8F4C	ACCB	00 7FFF FFFF	
CORCON	0040	CORCON	0040	
SR	0000	SR	1400	(SB, SAB=1)

SUBB

Subtract WREG and Carry bit from f

Syntax:	{label:}	SUBB{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (f) – (WREG) – ($\overline{C}$) → destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

1011	0101	1BDf	ffff	ffff	ffff

Description: Subtract the contents of the default working register WREG and the Borrow flag (Carry flag inverse, $\overline{C}$) from the contents of the specified file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, 1 for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The WREG is set to working register W0.
3: The Z flag is "sticky" for `ADDC, CPB, SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
SUBB.B 0x1FFF    ; Sub. WREG and C̅ from (0x1FFF) (Byte mode)
                 ; Store result to 0x1FFF
```

	Before Instruction		After Instruction	
WREG (W0)	7804	WREG (W0)	7804	
Data 1FFE	9439	Data 1FFE	8F39	
SR	0000	SR	0008	(N=1)

Example 2

```
SUBB 0xA04, WREG    ; Sub. WREG and C̅ from (0xA04) (Word mode)
                    ; Store result to WREG
```

	Before Instruction		After Instruction	
WREG (W0)	6234	WREG (W0)	0000	
Data 0A04	6235	Data 0A04	6235	
SR	0000	SR	0001	(C=1)

Advance Information

SUBB

Subtract Wn from Literal with Borrow

Syntax:	{label:} SUBB{.B} #lit10, Wn
Operands:	lit10 ∈ [0 ... 255] for byte operation lit10 ∈ [0 ... 1023] for word operation Wn ∈ [W0 ... W15]
Operation:	(Wn) – lit10 – ($\overline{C}$) → Wn
Status Affected:	DC, N, OV, Z, C

Encoding:

1011	0001	1Bkk	kkkk	kkkk	dddd

Description: Subtract the unsigned 10-bit literal operand and the Borrow flag (Carry flag inverse, $\overline{C}$) from the contents of the working register Wn, and store the result back in the working register Wn. Register direct addressing must be used for Wn.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'k' bits specify the literal operand.
The 'd' bits select the address of the working register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See **Section 4.6 "Using 10-bit Literal Operands"** for information on using 10-bit literal operands in Byte mode.

3: The Z flag is "sticky" for `ADDC, CPB, SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
SUBB.B  #0x23, W0    ; Sub. 0x23 and C̅ from W0 (Byte mode)
                     ; Store result to W0
```

	Before Instruction	After Instruction
W0	7804	78E0
SR	0000	0108 (DC, N=1)

Example 2

```
SUBB    #0x108, W4   ; Sub. 0x108 and C̅ from W4 (Word mode)
                     ; Store result to W4
```

	Before Instruction	After Instruction
W4	6234	612C
SR	0001 (C=1)	0001 (C=1)

SUBB

Subtract Short Literal from Wb with Borrow

Syntax:	{label:}	SUBB{.B}	Wb,	#lit5,	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[++Wd]
					[--Wd]

Operands: $Wb \in [W0 ... W15]$
$lit5 \in [0 ... 31]$
$Wd \in [W0 ... W15]$

Operation: $(Wb) - lit5 - (\overline{C}) \rightarrow Wd$

Status Affected: DC, N, OV, Z, C

Encoding:

0101	1www	wBqq	qddd	d11k	kkkk

Description: Subtract the 5-bit unsigned literal operand and the Borrow flag (Carry flag inverse, $\overline{C}$) from the contents of the base register Wb and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'k' bits provide the literal operand, a five-bit integer number.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The Z flag is "sticky" for `ADDC`, `CPB`, `SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
SUBB.B  W4, #0x10, W5   ; Sub. 0x10 and C̄ from W4 (Byte mode)
                        ; Store result to W5
```

	Before Instruction		After Instruction	
W4	1782	W4	1782	
W5	7804	W5	7871	
SR	0000	SR	0005	(OV, C=1)

Example 2

```
SUBB  W0, #0x8, [W2++]   ; Sub. 0x8 and C̄ from W0 (Word mode)
                         ; Store result to [W2]
                         ; Post-increment W2
```

	Before Instruction			After Instruction	
W0	0009		W0	0009	
W2	2004		W2	2006	
Data 2004	A557		Data 2004	0000	
SR	0020	(Z=1)	SR	0103	(DC, Z, C=1)

SUBB — Subtract Ws from Wb with Borrow

Syntax:	{label:}	SUBB{.B}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]
				[++Ws],	[++Wd]
				[--Ws],	[--Wd]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Wb) – (Ws) – ($\overline{C}$) → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0101	1www	wBqq	qddd	dppp	ssss

Description: Subtract the contents of the source register Ws and the Borrow flag (Carry flag inverse, $\overline{C}$) from the contents of the base register Wb, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The Z flag is “sticky” for `ADDC`, `CPB`, `SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
SUBB.B  W0, W1, W0    ; Sub. W1 and C̄ from W0 (Byte mode)
                      ; Store result to W0
```

	Before Instruction	After Instruction	
W0	1732	17ED	
W1	7844	7844	
SR	0000	0108	(DC, N=1)

Example 2

```
SUBB  W7,[W8++],[W9++]  ; Sub. [W8] and C̄ from W7 (Word mode)
                        ; Store result to [W9]
                        ; Post-increment W8
                        ; Post-increment W9
```

	Before Instruction		After Instruction	
W7	2450	W7	2450	
W8	1808	W8	180A	
W9	2022	W9	2024	
Data 1808	92E4	Data 1808	92E4	
Data 2022	A557	Data 2022	916C	
SR	0000	SR	010C	(DC, N, OV=1)

 Advance Information

SUBBR

Subtract f from WREG with Borrow

Syntax:	{label:}	SUBBR{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (WREG) – (f) – ($\overline{C}$) → destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

1011	1101	1BDf	ffff	ffff	ffff

Description: Subtract the contents of the specified file register f and the Borrow flag (Carry flag inverse, $\overline{C}$) from the contents of WREG, and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

> **Note 1:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
> **2:** The WREG is set to working register W0.
> **3:** The Z flag is "sticky" for `ADDC, CPB, SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
SUBBR.B 0x803  ; Sub. (0x803) and C̄ from WREG (Byte mode)
               ; Store result to 0x803
```

	Before Instruction			After Instruction
WREG (W0)	7804		WREG (W0)	7804
Data 0802	9439		Data 0802	6F39
SR	0002	(Z=1)	SR	0000

Example 2

```
SUBBR 0xA04, WREG ; Sub. (0xA04) and C̄ from WREG (Word mode)
                  ; Store result to WREG
```

	Before Instruction		After Instruction	
WREG (W0)	6234	WREG (W0)	FFFE	
Data 0A04	6235	Data 0A04	6235	
SR	0000	SR	0008	(N=1)

5

Instruction Descriptions

SUBBR

Subtract Wb from Short Literal with Borrow

Syntax:	{label:}	SUBBR{.B}	Wb,	#lit5,	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[++Wd]
					[--Wd]

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
Wd ∈ [W0 ... W15]

Operation: lit5 – (Wb) – ($\overline{C}$) → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0001	1www	wBqq	qddd	d11k	kkkk

Description: Subtract the contents of the base register Wb and the Borrow flag (Carry flag inverse, $\overline{C}$) from the 5-bit unsigned literal and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing must be used for Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'k' bits provide the literal operand, a five-bit integer number.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.
2: The Z flag is "sticky" for `ADDC, CPB, SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
SUBBR.B  W0, #0x10, W1 ; Sub. W0 and C̄ from 0x10 (Byte mode)
                       ; Store result to W1
```

	Before Instruction			After Instruction	
W0	F310		W0	F310	
W1	786A		W1	7800	
SR	0003	(Z, C=1)	SR	0103	(DC, Z, C=1)

Example 2

```
SUBBR  W0, #0x8, [W2++] ; Sub. W0 and C̄ from 0x8 (Word mode)
                        ; Store result to [W2]
                        ; Post-increment W2
```

	Before Instruction			After Instruction	
W0	0009		W0	0009	
W2	2004		W2	2006	
Data 2004	A557		Data 2004	FFFE	
SR	0020	(Z=1)	SR	0108	(DC, N=1)

SUBBR

Subtract Wb from Ws with Borrow

Syntax:	{label:}	SUBBR{.B} Wb,	Ws,	Wd
			[Ws],	[Wd]
			[Ws++],	[Wd++]
			[Ws--],	[Wd--]
			[++Ws],	[++Wd]
			[--Ws],	[--Wd]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Ws) – (Wb) – ($\overline{C}$) → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0001	1www	wBqq	qddd	dppp	ssss

Description: Subtract the contents of the base register Wb and the Borrow flag (Carry flag inverse, $\overline{C}$) from the contents of the source register Ws and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The Z flag is “sticky” for `ADDC, CPB, SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Example 1

```
SUBBR.B  W0, W1, W0    ; Sub. W0 and C̅ from W1 (Byte mode)
                       ; Store result to W0
```

	Before Instruction		After Instruction	
W0	1732	W0	1711	
W1	7844	W1	7844	
SR	0000	SR	0001	(C=1)

5

Instruction Descriptions

Example 2

```
SUBBR W7,[W8++],[W9++] ; Sub. W7 and C̅ from [W8] (Word mode)
                       ; Store result to [W9]
                       ; Post-increment W8
                       ; Post-increment W9
```

	Before Instruction	After Instruction	
W7	2450	2450	
W8	1808	180A	
W9	2022	2024	
Data 1808	92E4	92E4	
Data 2022	A557	6E93	
SR	0000	0005	(OV, C=1)

SUBR

Subtract f from WREG

Syntax:	{label:}	SUBR{.B}	f	{,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (WREG) – (f) → destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

1011	1101	0BDf	ffff	ffff	ffff

Description: Subtract the contents of the specified file register from the contents of the default working register WREG, and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'D' bit selects the destination (`0` for WREG, `1` for file register).
The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1

```
SUBR.B  0x1FFF   ; Sub. (0x1FFF) from WREG (Byte mode)
                 ; Store result to 0x1FFF
```

	Before Instruction		After Instruction
WREG (W0)	7804	WREG (W0)	7804
Data 1FFE	9439	Data 1FFE	7039
SR	0000	SR	0000

Example 2

```
SUBR   0xA04, WREG   ; Sub. (0xA04) from WREG (Word mode)
                     ; Store result to WREG
```

	Before Instruction		After Instruction
WREG (W0)	6234	WREG (W0)	FFFF
Data 0A04	6235	Data 0A04	6235
SR	0000	SR	0008 (N=1)

5

Instruction Descriptions

SUBR — Subtract Wb from Short Literal

Syntax:	{label:}	SUBR{.B}	Wb,	#lit5	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[++Wd]
					[--Wd]

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
Wd ∈ [W0 ... W15]

Operation: lit5 – (Wb) → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0001	0www	wBqq	qddd	d11k	kkkk

Description: Subtract the contents of the base register Wb from the unsigned 5-bit literal operand, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'k' bits provide the literal operand, a five-bit integer number.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
SUBR.B  W0, #0x10, W1   ; Sub. W0 from 0x10 (Byte mode)
                        ; Store result to W1
```

	Before Instruction	After Instruction	
W0	F310	F310	
W1	786A	7800	
SR	0000	0103	(DC, Z, C=1)

Example 2

```
SUBR   W0, #0x8, [W2++]   ; Sub. W0 from 0x8 (Word mode)
                          ; Store result to [W2]
                          ; Post-increment W2
```

	Before Instruction	After Instruction	
W0	0009	0009	
W2	2004	2006	
Data 2004	A557	FFFF	
SR	0000	0108	(DC, N=1)

 Advance Information

SUBR

Subtract Wb from Ws

Syntax:	{label:}	SUBR{.B}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]
				[++Ws],	[++Wd]
				[--Ws],	[--Wd]

Operands: Wb $\in$ [W0 ... W15]
Ws $\in$ [W0 ... W15]
Wd $\in$ [W0 ... W15]

Operation: (Ws) – (Wb) $\rightarrow$ Wd

Status Affected: DC, N, OV, Z, C

Encoding:

0001	0www	wBqq	qddd	dppp	ssss

Description: Subtract the contents of the base register Wb from the contents of the source register Ws and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
SUBR.B  W0, W1, W0   ; Sub. W0 from W1 (Byte mode)
                     ; Store result to W0
```

	Before Instruction		After Instruction	
W0	1732	W0	1712	
W1	7844	W1	7844	
SR	0000	SR	0001	(C=1)

Example 2

```
SUBR  W7, [W8++], [W9++]   ; Sub. W7 from [W8] (Word mode)
                           ; Store result to [W9]
                           ; Post-increment W8
                           ; Post-increment W9
```

	Before Instruction		After Instruction	
W7	2450	W7	2450	
W8	1808	W8	180A	
W9	2022	W9	2024	
Data 1808	92E4	Data 1808	92E4	
Data 2022	A557	Data 2022	6E94	
SR	0000	SR	0005	(OV, C=1)

SWAP

Byte or Nibble Swap Wn

Syntax:	{label:}	SWAP{.B} Wn

Operands: Wn ∈ [W0 ... W15]

Operation:
For byte operation:
(Wn)<7:4> ↔ (Wn)<3:0>
For word operation:
(Wn)<15:8> ↔ (Wn)<7:0>

Status Affected: None

Encoding:

1111	1101	1B00	0000	0000	ssss

Description: Swap the contents of the working register Wn. In Word mode, the two bytes of Wn are swapped. In Byte mode, the two nibbles of the Least Significant Byte of Wn are swapped, and the Most Significant Byte of Wn is unchanged. Register direct addressing must be used for Wn.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘s’ bits select the address of the working register.

Note: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1 `SWAP.B  0x1021    ; Nibble swap (0x1021)`

	Before Instruction		After Instruction
Data 1020	AB87	Data 1020	BA87
SR	0000	SR	0000

Example 2 `SWAP    0x1760    ; Byte swap (0x1760)`

	Before Instruction		After Instruction
Data 1760	8095	Data 1760	9580
SR	0000	SR	0000

TBLRDH **Table Read High**

Syntax:	{label:}	TBLRDH{.B}	[Ws],	Wd
			[Ws++],	[Wd]
			[Ws--],	[Wd++]
			[++Ws],	[Wd--]
			[--Ws],	[++Wd]
				[--Wd]

Operands: Ws $\in$ [W0 ... W15]
Wd $\in$ [W0 ... W15]

Operation: <u>For byte operation:</u>
If (LSB(Ws) = `1`)
`0` $\rightarrow$ Wd
Else
Program Mem [(TBLPAG),(Ws)] <23:16> $\rightarrow$ Wd
<u>For word operation:</u>
Program Mem [(TBLPAG),(Ws)] <23:16> $\rightarrow$ Wd <7:0>
`0` $\rightarrow$ Wd <15:8>

Status Affected: None

Encoding:

1011	1010	1Bqq	qddd	dppp	ssss

Description: Read the contents of the Most Significant Word of program memory and store it to the destination register Wd. The target word address of program memory is formed by concatenating the 8-bit Table Pointer register, TBLPAG<7:0>, with the effective address specified by Ws. Indirect addressing must be used for Ws, and either register direct or indirect addressing may be used for Wd.

In Word mode, zero is stored to the Most Significant Byte of the destination register (due to non-existent program memory) and the third program memory byte (PM<23:16>) at the specified program memory address is stored to the Least Significant Byte of the destination register.

In Byte mode, the source address depends on the contents of Ws. If Ws is not word aligned, zero is stored to the destination register (due to non-existent program memory). If Ws is word aligned, the third program memory byte (PM<23:16>) at the specified program memory address is stored to the destination register.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination (data) register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the source (address) register.

Note: The extension `.B` in the instruction denotes a byte move rather than a word move. You may use a `.W` extension to denote a word move, but it is not required.

Words: 1

Cycles: 2

Example 1

```
TBLRDH.B  [W0], [W1++]   ; Read PM (TBLPAG:[W0]) (Byte mode)
                         ; Store to [W1]
                         ; Post-increment W1
```

	Before Instruction		After Instruction
W0	0812	W0	0812
W1	0F71	W1	0F72
Data 0F70	0944	Data 0F70	EF44
Program 01 0812	EF 2042	Program 01 0812	EF 2042
TBLPAG	0001	TBLPAG	0001
SR	0000	SR	0000

Example 2

```
TBLRDH   [W6++], W8   ; Read PM (TBLPAG:[W6]) (Word mode)
                      ; Store to W8
                      ; Post-increment W6
```

	Before Instruction		After Instruction
W6	3406	W6	3408
W8	65B1	W8	0029
Program 00 3406	29 2E40	Program 00 3406	29 2E40
TBLPAG	0000	TBLPAG	0000
SR	0000	SR	0000

TBLRDL Table Read Low

Syntax:	{label:}	TBLRDL{.B}	[Ws],	Wd
			[Ws++],	[Wd]
			[Ws--],	[Wd++]
			[++Ws],	[Wd--]
			[--Ws],	[++Wd]
				[--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: For byte operation:
If (LSB(Ws) = 1)
Program Mem [(TBLPAG),(Ws)] <15:8> → Wd
Else
Program Mem [(TBLPAG),(Ws)] <7:0> → Wd
For word operation:
Program Mem [(TBLPAG),(Ws)] <15:0> → Wd

Status Affected: None

Encoding:

1011	1010	0Bqq	qddd	dppp	ssss

Description: Read the contents of the Least Significant Word of program memory and store it to the destination register Wd. The target word address of program memory is formed by concatenating the 8-bit Table Pointer register, TBLPAG<7:0>, with the effective address specified by Ws. Indirect addressing must be used for Ws, and either register direct or indirect addressing may be used for Wd.

In Word mode, the lower 2 bytes of program memory are stored to the destination register. In Byte mode, the source address depends on the contents of Ws. If Ws is not word aligned, the second byte of the program memory word (PM<15:7>) is stored to the destination register. If Ws is word aligned, the first byte of the program memory word (PM<7:0>) is stored to the destination register.

The 'B' bit selects byte or word operation (0 for word mode, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination (data) register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source (address) register.

Note: The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

Words: 1

Cycles: 2

Example 1

```
TBLRDL.B  [W0++], W1   ; Read PM (TBLPAG:[W0]) (Byte mode)
                       ; Store to W1
                       ; Post-increment W0
```

	Before Instruction		After Instruction
W0	0813	W0	0814
W1	0F71	W1	0F20
Data 0F70	0944	Data 0F70	EF44
Program 01 0812	EF 2042	Program 01 0812	EF 2042
TBLPAG	0001	TBLPAG	0001
SR	0000	SR	0000

Example 2

```
TBLRDL    [W6], [W8++]   ; Read PM (TBLPAG:[W6]) (Word mode)
                         ; Store to W8
                         ; Post-increment W8
```

	Before Instruction		After Instruction
W6	3406	W6	3408
W8	1202	W8	1204
Data 1202	658B	Data 1202	2E40
Program 00 3406	29 2E40	Program 00 3406	29 2E40
TBLPAG	0000	TBLPAG	0000
SR	0000	SR	0000

TBLWTH — Table Write High

Syntax:	{label:}	TBLWTH{.B}	Ws,	[Wd]
			[Ws],	[Wd++]
			[Ws++],	[Wd--]
			[Ws--],	[++Wd]
			[++Ws],	[--Wd]
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation:
For byte operation:
If (LSB(Wd) = 1)
NOP
Else
(Ws) → Program Mem [(TBLPAG),(Wd)]<23:16>
For word operation:
(Ws)<7:0> → Program Mem [(TBLPAG),(Wd)] <23:16>

Status Affected: None

Encoding:

1011	1011	1Bqq	qddd	dppp	ssss

Description: Store the contents of the working source register Ws to the Most Significant Word of program memory. The destination word address of program memory is formed by concatenating the 8-bit Table Pointer register, TBLPAG<7:0>, with the effective address specified by Wd. Either direct or indirect addressing may be used for Ws, and indirect addressing must be used for Wd.

Since program memory is 24-bits wide, this instruction can only write to the upper byte of program memory (PM<23:16>). This may be performed using a Wd that is word aligned in Byte mode or Word mode. If Byte mode is used with a Wd that is not word aligned, no operation is performed.

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination (address) register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source (data) register.

Note: The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

Words: 1

Cycles: 2

Example 1

```
TBLWTH.B  [W0++], [W1]   ; Write [W0]... (Byte mode)
                         ; to PM Latch High (TBLPAG:[W1])
                         ; Post-increment W0
```

	Before Instruction		After Instruction
W0	0812	W0	0812
W1	0F70	W1	0F70
Data 0812	0944	Data 0812	EF44
Program 01 0F70	EF 2042	Program 01 0F70	44 2042
TBLPAG	0001	TBLPAG	0001
SR	0000	SR	0000

Note: Only the Program Latch is written to. The contents of program memory are not updated until the FLASH memory is programmed using the procedure described in the dsPIC30F Family Reference Manual.

Example 2

```
TBLWTH   W6, [W8++]     ; Write W6... (Word mode)
                        ; to PM Latch High (TBLPAG:[W8])
                        ; Post-increment W8
```

	Before Instruction		After Instruction
W6	0026	W6	0026
W8	0870	W8	0872
Program 00 0870	22 3551	Program 00 0870	26 3551
TBLPAG	0000	TBLPAG	0000
SR	0000	SR	0000

Note: Only the Program Latch is written to. The contents of program memory are not updated until the FLASH memory is programmed using the procedure described in the dsPIC30F Family Reference Manual.

TBLWTL — Table Write Low

Syntax:	{label:}	TBLWTL{.B}	Ws,	[Wd]
			[Ws],	[Wd++]
			[Ws++],	[Wd--]
			[Ws--],	[++Wd]
			[++Ws],	[--Wd]
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: For byte operation:
If (LSB(Wd)=1)
(Ws) → Program Mem [(TBLPAG),(Wd)] <15:8>
Else
(Ws) → Program Mem [(TBLPAG),(Wd)] <7:0>
For word operation:
(Ws) → Program Mem [(TBLPAG),(Wd)] <15:0>

Status Affected: None

Encoding:

1011	1011	0Bqq	qddd	dppp	ssss

Description: Store the contents of the working source register Ws to the Least Significant Word of program memory. The destination word address of program memory is formed by concatenating the 8-bit Table Pointer register, TBLPAG<7:0>, with the effective address specified by Wd. Either direct or indirect addressing may be used for Ws, and indirect addressing must be used for Wd.

In Word mode, Ws is stored to the lower 2 bytes of program memory. In Byte mode, the Least Significant bit of Wd determines the destination byte. If Wd is not word aligned, Ws is stored to the second byte of program memory (PM<15:8>). If Wd is word aligned, Ws is stored to the first byte of program memory (PM<7:0>).

The 'B' bit selects byte or word operation (0 for word, 1 for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination (address) register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source (data) register.

Note: The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

Words: 1

Cycles: 2

Example 1

```
TBLWTL.B  W0, [W1++]   ; Write W0... (Byte mode)
                       ; to PM Latch Low (TBLPAG:[W1])
                       ; Post-increment W1
```

	Before Instruction		After Instruction
W0	6628	W0	6628
W1	1225	W1	1226
Program 00 1224	78 0080	Program 01 1224	78 2880
TBLPAG	0000	TBLPAG	0000
SR	0000	SR	0000

Note: Only the Program Latch is written to. The contents of program memory are not updated until the FLASH memory is programmed using the procedure described in the dsPIC30F Family Reference Manual.

Example 2

```
TBLWTL    [W6], [W8]   ; Write [W6]... (Word mode)
                       ; to PM Latch Low (TBLPAG:[W8])
                       ; Post-increment W8
```

	Before Instruction		After Instruction
W6	1600	W6	1600
W8	7208	W8	7208
Data 1600	0130	Data 1600	0130
Program 01 7208	09 0002	Program 01 7208	09 0130
TBLPAG	0001	TBLPAG	0001
SR	0000	SR	0000

Note: Only the Program Latch is written to. The contents of program memory are not updated until the FLASH memory is programmed using the procedure described in the dsPIC30F Family Reference Manual.

ULNK

De-allocate Stack Frame

Syntax:	{label:} ULNK
Operands:	None
Operation:	W14 → W15 (W15)-2 → W15 (TOS) → W14
Status Affected:	None
Encoding:	1111 1010 1000 0000 0000 0000
Description:	This instruction de-allocates a stack frame for a subroutine calling sequence. The stack frame is de-allocated by setting the stack pointer (W15) equal to the frame pointer (W14), and then popping the stack to reset the frame pointer (W14).
Words:	1
Cycles:	1

Example 1

```
ULNK    ; Unlink the stack frame
```

	Before Instruction		After Instruction
W14	2002	W14	2000
W15	20A2	W15	2000
Data 2000	2000	Data 2000	2000
SR	0000	SR	0000

Example 2

```
ULNK    ; Unlink the stack frame
```

	Before Instruction		After Instruction
W14	0802	W14	0800
W15	0812	W15	0800
Data 0800	0800	Data 0800	0800
SR	0000	SR	0000

XOR

Exclusive OR f and WREG

Syntax:	{label:} XOR{.B} f {,WREG}
Operands:	f ∈ [0 ... 8191]
Operation:	(f).XOR.(WREG) → destination designated by D
Status Affected:	N, Z
Encoding:	`1011` `0110` `1BDf` `ffff` `ffff` `ffff`
Description:	Compute the logical exclusive OR operation of the contents of the default working register WREG and the contents of the specified file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (`0` for word, `1` for byte).
The ‘D’ bit selects the destination (`0` for WREG, `1` for file register).
The ‘f’ bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words:	1
Cycles:	1

Example 1
```
XOR.B   0x1FFF    ; XOR (0x1FFF) and WREG (Byte mode)
                  ; Store result to 0x1FFF
```

	Before Instruction		After Instruction	
WREG (W0)	7804	WREG (W0)	7804	
Data 1FFE	9439	Data 1FFE	9039	
SR	0000	SR	0008	(N=1)

Example 2
```
XOR     0xA04, WREG  ; XOR (0xA04) and WREG (Word mode)
                     ; Store result to WREG
```

	Before Instruction		After Instruction	
WREG (W0)	6234	WREG (W0)	C267	
Data 0A04	A053	Data 0A04	A053	
SR	0000	SR	0008	(N=1)

XOR

Exclusive OR Literal and Wn

Syntax:	{label:}	XOR{.B}	#lit10,	Wn		

Operands: lit10 ∈ [0 ... 255] for byte operation
lit10 ∈ [0 ... 1023] for word operation
Wn ∈ [W0 ... W15]

Operation: lit10.XOR.(Wn) → Wn

Status Affected: N, Z

Encoding:

1011	0010	1Bkk	kkkk	kkkk	dddd

Description: Compute the logical exclusive OR operation of the unsigned 10-bit literal operand and the contents of the working register Wn and store the result back in the working register Wn. Register direct addressing must be used for Wn.

The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'k' bits specify the literal operand.
The 'd' bits select the address of the working register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See **Section 4.6 "Using 10-bit Literal Operands"** for information on using 10-bit literal operands in Byte mode.

Words: 1

Cycles: 1

Example 1

```
XOR.B  #0x23, W0     ; XOR 0x23 and W0 (Byte mode)
                     ; Store result to W0
```

	Before Instruction	After Instruction
W0	7804	7827
SR	0000	0000

Example 2

```
XOR   #0x108, W4     ; XOR 0x108 and W4 (Word mode)
                     ; Store result to W4
```

	Before Instruction	After Instruction
W4	6134	603C
SR	0000	0000

XOR

Exclusive OR Wb and Short Literal

Syntax:	{label:}	XOR{.B}	Wb,	#lit5,	Wd
					[Wd]
					[Wd++]
					[Wd--]
					[++Wd]
					[--Wd]

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
Wd ∈ [W0 ... W15]

Operation: (Wb).XOR.lit5 → Wd

Status Affected: N, Z

Encoding:

0110	1www	wBqq	qddd	d11k	kkkk

Description: Compute the logical exclusive OR operation of the contents of the base register Wb and the unsigned 5-bit literal operand and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (0 for word, 1 for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the address of the destination register.
The ‘k’ bits provide the literal operand, a 5-bit integer number.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
XOR.B  W4, #0x16, W5     ; XOR W4 and 0x14 (Byte mode)
                         ; Store result to W5
```

	Before Instruction	After Instruction
W4	C822	C822
W5	1200	1234
SR	0000	0000

Example 2

```
XOR    W2, #0x1F, [W8++]  ; XOR W2 by 0x1F (Word mode)
                          ; Store result to [W8]
                          ; Post-increment W8
```

	Before Instruction	After Instruction
W2	8505	8505
W8	1004	1006
Data 1004	6628	851A
SR	0000	0008 (N=1)

XOR

Exclusive OR Wb and Ws

Syntax:	{label:}	XOR{.B}	Wb,	Ws,	Wd
				[Ws],	[Wd]
				[Ws++],	[Wd++]
				[Ws--],	[Wd--]
				[++Ws],	[++Wd]
				[--Ws],	[--Wd]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Wb).XOR.(Ws) → Wd

Status Affected: N, Z

Encoding:

0110	1www	wBqq	qddd	dppp	ssss

Description: Compute the logical exclusive OR operation of the contents of the source register Ws and the contents of the base register Wb, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation (`0` for word, `1` for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

> **Note:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1

```
XOR.B  W1, [W5++], [W9++]  ; XOR W1 and [W5] (Byte mode)
                           ; Store result to [W9]
                           ; Post-increment W5 and W9
```

	Before Instruction		After Instruction
W1	AAAA	W1	AAAA
W5	2000	W5	2001
W9	2600	W9	2601
Data 2000	115A	Data 2000	115A
Data 2600	0000	Data 2600	00F0
SR	0000	SR	0008 (N=1)

Example 2

```
XOR  W1, W5, W9          ; XOR W1 and W5 (Word mode)
                         ; Store the result to W9
```

	Before Instruction	After Instruction	
W1	FEDC	FEDC	
W5	1234	1234	
W9	A34D	ECE8	
SR	0000	0008	(N=1)

ZE — Zero-Extend Wn

Syntax:	{label:}	ZE	Ws,	Wnd
			[Ws],	
			[Ws++],	
			[Ws--],	
			[++Ws],	
			[--Ws],	

Operands: Ws ∈ [W0 ... W15]
Wnd ∈ [W0 ... W15]

Operation: Ws<7:0> → Wnd<7:0>
0 → Wnd<15:8>

Status Affected: N, Z, C

Encoding:

1111	1011	10qq	qddd	dppp	ssss

Description: Zero-extend the Least Significant Byte in source working register Ws to a 16-bit value and store the result in the destination working register Wnd. Either register direct or indirect addressing may be used for Ws, and register direct addressing must be used for Wnd. The N flag is cleared and the C flag is set, because the zero-extended word is always positive.

The 'q' bits select the destination Address mode.
The 'd' bits select the address of the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the address of the source register.

Note 1: This operation converts a byte to a word, and it uses no `.B` or `.W` extension.

2: The source Ws is addressed as a byte operand, so any address modification is by `1`.

Words: 1

Cycles: 1

Example 1

```
ZE   W3, W4    ; zero-extend W3
               ; Store result to W4
```

	Before Instruction	After Instruction	
W3	7839	7839	
W4	1005	0039	
SR	0000	0001	(C=1)

Example 2

```
ZE   [W2++], W12   ; Zero-extend [W2]
                   ; Store to W12
                   ; Post-increment W2
```

	Before Instruction	After Instruction	
W2	0900	0901	
W12	1002	008F	
Data 0900	268F	268F	
SR	0000	0001	(C=1)

 Advance Information

Section 6. Reference

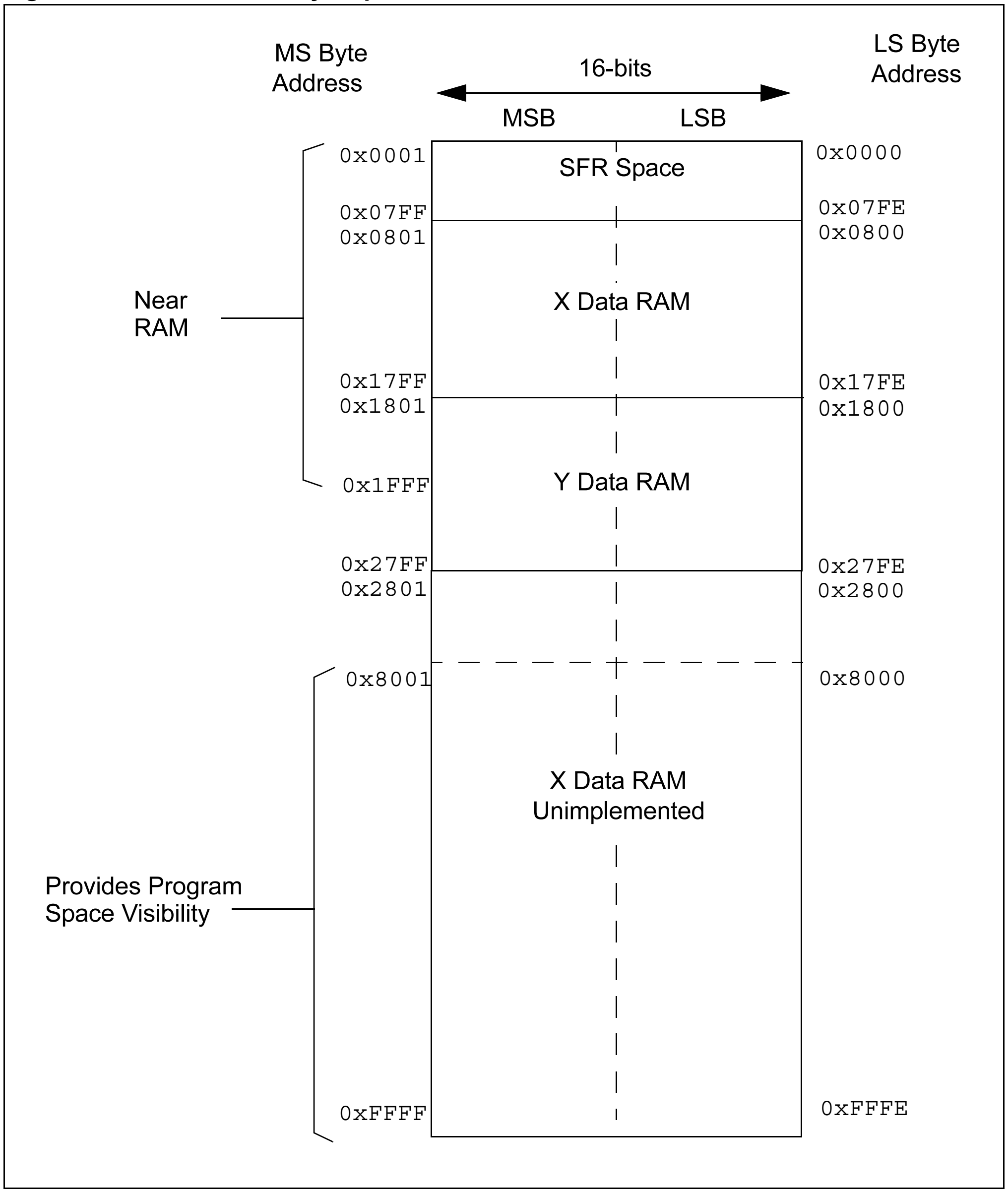
HIGHLIGHTS

This section of the manual contains reference information for the dsPIC30F. It consists of the following sections:

6.1 Data Memory Map

A sample dsPIC30F data memory map is shown in Figure 6-1.

Figure 6-1: Data Memory Map



Note 1:	The partition between the X and Y data spaces is device specific. Refer to the appropriate device data sheet for further details. The data space boundaries indicated here are for example purposes only.
2:	Refer to **Section 4. "Instruction Set Details"** for information on Data Addressing modes, performing byte accesses and word alignment requirements.
3:	Refer to the dsPIC30F MCU Family Reference Manual for information on accessing program memory through data address space.

6.2 Core Special Function Register Map

The Core Special Function Register Map is shown in Table 6-1. Please refer to the dsPIC30F Data Sheet for complete register descriptions and the memory map of the remaining special function registers.

Table 6-1: dsPIC30F Core Register Map

Name	Addr	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RESET State
W0	0000	W0 (WREG)																0000 0000 0000 0000
W1	0002	W1																0000 0000 0000 0000
W2	0004	W2																0000 0000 0000 0000
W3	0006	W3																0000 0000 0000 0000
W4	0008	W4																0000 0000 0000 0000
W5	000A	W5																0000 0000 0000 0000
W6	000C	W6																0000 0000 0000 0000
W7	000E	W7																0000 0000 0000 0000
W8	0010	W8																0000 0000 0000 0000
W9	0012	W9																0000 0000 0000 0000
W10	0014	W10																0000 0000 0000 0000
W11	0016	W11																0000 0000 0000 0000
W12	0018	W12																0000 0000 0000 0000
W13	001A	W13																0000 0000 0000 0000
W14	001C	W14																0000 0000 0000 0000
W15	001E	W15																0000 1000 0000 0000
SPLIM	0020	SPLIM																0000 0000 0000 0000
ACCAL	0022	ACCAL																0000 0000 0000 0000
ACCAH	0024	ACCAH																0000 0000 0000 0000
ACCAU	0026	Sign-extension of ACCA<39>								ACCAU								0000 0000 0000 0000
ACCBL	0028	ACCBL																0000 0000 0000 0000
ACCBH	002A	ACCBH																0000 0000 0000 0000
ACCBU	002C	Sign-extension of ACCB<39>								ACCBU								0000 0000 0000 0000
PCL	002E	PCL																0000 0000 0000 0000
PCH	0030	—	—	—	—	—	—	—	—	—	PCH							0000 0000 0000 0000
TBLPAG	0032	—	—	—	—	—	—	—	—	TBLPAG								0000 0000 0000 0000
PSVPAG	0034	—	—	—	—	—	—	—	—	PSVPAG								0000 0000 0000 0000
RCOUNT	0036	RCOUNT																xxxx xxxx xxxx xxxx
DCOUNT	0038	DCOUNT																xxxx xxxx xxxx xxxx
DOSTARTL	003A	DOSTARTL																xxxx xxxx xxxx xxxx
DOSTARTH	003C	—	—	—	—	—	—	—	—	—	—	DOSTARTH						0000 0000 00xx xxxx
DOENDL	003E	DOENDL																xxxx xxxx xxxx xxxx
DOENDH	0040	—	—	—	—	—	—	—	—	—	—	DOENDH						0000 0000 00xx xxxx

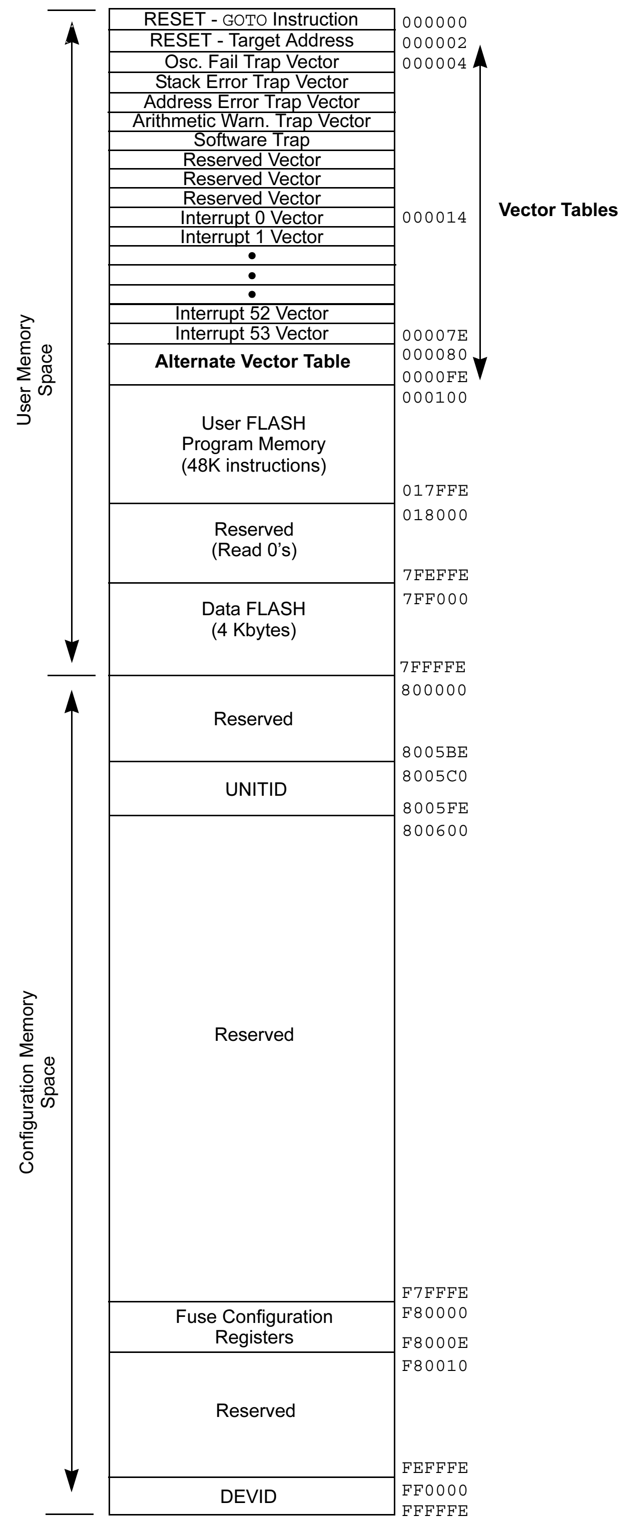
Table 6-1: dsPIC30F Core Register Map (Continued)

Name	Addr	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	RESET State
SR	0042	OA	OB	SA	SB	OAB	SAB	DA	DC	IPL2	IPL1	IPL0	RA	N	OV	Z	C	0000 0000 0000 0000
CORCON	0044	—	—	—	US	EDT	DL2	DL1	DL0	SATA	SATB	SATDW	ACCSAT	IPL3	PSV	RND	IF	0000 0000 0010 0000
MODCON	0046	XMODEN	YMODEN	—	—	BWM<3:0>				YWM<3:0>				XWM<3:0>				0000 0000 0000 0000
XMODSRT	0048	XMODSRT<15:0>																xxxx xxxx xxxx xxxx
XMODEND	004A	XMODEND<15:0>																xxxx xxxx xxxx xxxx
YMODSRT	004C	YMODSRT<15:0>																xxxx xxxx xxxx xxxx
YMODEND	004E	YMODEND<15:0>																xxxx xxxx xxxx xxxx
XBREV	0050	BREN	XBREV<14:0>															xxxx xxxx xxxx xxxx
DISICNT	0052	—	—	DISICNT<13:0>														0000 0000 0000 0000
Reserved	0054 - 007E	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000 0000 0000 0000

6.3 Program Memory Map

A sample dsPIC30F program memory map is shown in Figure 6-2.

Figure 6-2: Program Space Memory Map



6.4 Instruction Bit Map

Instruction encoding for the dsPIC30F is summarized in Table 6-2. This table contains the encoding for the Most Significant Byte of each instruction. The first column in the table represents bits 23:20 of the opcode, and the first row of the table represents bits 19:16 of the opcode. The first byte of the opcode is formed by taking the first column bit value and appending the first row bit value. For instance, the Most Significant Byte of the `PUSH` instruction (last row, ninth column) is encoded with `11111000b (0xF8)`.

Note:	The complete opcode for each instruction may be determined by the instruction descriptions in **Section 5. "Instruction Descriptions"**, using Table 5.2 through Table 5-12.

Table 6-2: dsPIC30F Instruction Encoding

bit 23 bit 19	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NOP	BRA CALL GOTO RCALL	CALL	—	GOTO	RETLW	RETFIE RETURN	RCALL	DO	REPEAT	—	—	BRA (OA)	BRA (OB)	BRA (SA)	BRA (SB)
0001	SUBR								SUBBR							
0010	MOV															
0011	BRA (OV)	BRA (C)	BRA (Z)	BRA (N)	BRA (LE)	BRA (LT)	BRA (LEU)	BRA	BRA (NOV)	BRA (NC)	BRA (NZ)	BRA (NN)	BRA (GT)	BRA (GE)	BRA (GTU)	—
0100	ADD								ADDC							
0101	SUB								SUBB							
0110	AND								XOR							
0111	IOR								MOV							
1000	MOV															
1001	MOV															
1010	BSET	BCLR	BTG	BTST	BTSTS	BTST	BTSS	BTSC	BSET	BCLR	BTG	BTST	BTSTS	BSW	BTSS	BTSC
1011	ADD ADDC	SUB SUBB	AND XOR	IOR MOV	ADD ADDC	SUB SUBB	AND XOR	IOR MOV	MUL.US MUL.UU	MUL.SS MUL.SU	TBLRDH TBLRDL	TBLWTH TBLWTL	MUL	SUB SUBB	MOV.D	MOV
1100	MAC MPY MPY.N MSC			CLRAC	MAC MPY MPY.N MSC			MOVSAC	SFTAC	ADD	LAC	ADD NEG SUB	SAC	SAC.R	—	FF1L FF1R
1101	SL	ASR LSR	RLC RLNC	RRC RRNC	SL	ASR LSR	RLC RLNC	RRC RRNC	DIV.S DIV.U	DIVF	—	—	—	SL	ASR LSR	FBCL
1110	CP0	CP CPB	CP0	CP CPB	—	—	CPSGT CPSLT	CPSEQ CPSNE	INC INC2	DEC DEC2	COM NEG	CLR SETM	INC INC2	DEC DEC2	COM NEG	CLR SETM
1111	ED EDAC MAC MPY				—	—	—	—	PUSH	POP	LNK ULNK	SE ZE	DISI	DAW EXCH SWAP	CLRWDT PWRSAV POP.S PUSH.S RESET	NOPR

Advance Information

6.5 Instruction Set Summary Table

The complete dsPIC30F instruction set is summarized in Table 6-3. This table contains an alphabetized listing of the instruction set. It includes instruction assembly syntax, description, size (in 24-bit words), execution time (in instruction cycles), affected status bits and the page number in which the detailed description can be found. Table 1-2 identifies the symbols which are used in the Instruction Set Summary Table.

Table 6-3: dsPIC30F Instruction Set Summary Table

Assembly Syntax Mnemonic,Operands		Description	Words	Cycles	OA	OB	SA	SB	OAB	SAB	DC	N	OV	Z	C	Page #
ADD	f {,WREG}	Destination = f + WREG	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-7
ADD	#lit10,Wn	Wn = lit10 + Wn	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-8
ADD	Wb,#lit5,Wd	Wd = Wb + lit5	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-9
ADD	Wb,Ws,Wd	Wd = Wb + Ws	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-10
ADD	Acc	Add accumulators	1	1	⇳	⇳	⇧	⇧	⇳	⇧	—	—	—	—	—	5-11
ADD	Ws,#Slit4,Acc	16-bit signed add to accumulator	1	1	⇳	⇳	⇧	⇧	⇳	⇧	—	—	—	—	—	5-12
ADDC	f {,WREG}	Destination = f + WREG + (C)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇩	⇳	5-14
ADDC	#lit10,Wn	Wn = lit10 + Wn + (C)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇩	⇳	5-15
ADDC	Wb,#lit5,Wd	Wd = Wb + lit5 + (C)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇩	⇳	5-16
ADDC	Wb,Ws,Wd	Wd = Wb + Ws + (C)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇩	⇳	5-17
AND	f {,WREG}	Destination = f .AND. WREG	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-19
AND	#lit10,Wn	Wn = lit10 .AND. Wn	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-20
AND	Wb,#lit5,Wd	Wd = Wb .AND. lit5	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-21
AND	Wb,Ws,Wd	Wd = Wb .AND. Ws	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-22
ASR	f {,WREG}	Destination = arithmetic right shift f	1	1	—	—	—	—	—	—	—	⇳	—	⇳	⇳	5-24
ASR	Ws,Wd	Wd = arithmetic right shift Ws	1	1	—	—	—	—	—	—	—	⇳	—	⇳	⇳	5-25
ASR	Wb,#lit4,Wnd	Wnd = arithmetic right shift Wb by lit4	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-27
ASR	Wb,Wns,Wnd	Wnd = arithmetic right shift Wb by Wns	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-28
BCLR	f,#bit4	Bit clear f	1	1	—	—	—	—	—	—	—	—	—	—	—	5-29
BCLR	Ws,#bit4	Bit clear Ws	1	1	—	—	—	—	—	—	—	—	—	—	—	5-30
BRA	Expr	Branch unconditionally	1	2	—	—	—	—	—	—	—	—	—	—	—	5-31
BRA	Wn	Computed branch	1	2	—	—	—	—	—	—	—	—	—	—	—	5-32
BRA	C,Expr	Branch if Carry	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-33
BRA	GE,Expr	Branch if greater than or equal	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-35
BRA	GEU,Expr	Branch if Carry	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-36
BRA	GT,Expr	Branch if greater than	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-37
BRA	GTU,Expr	Branch if unsigned greater than	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-38
BRA	LE,Expr	Branch if less than or equal	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-39
BRA	LEU,Expr	Branch if unsigned less than or equal	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-40
BRA	LT,Expr	Branch if less than	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-41
BRA	LTU,Expr	Branch if not Carry	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-42
BRA	N,Expr	Branch if Negative	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-43
BRA	NC,Expr	Branch if not Carry	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-44
BRA	NN,Expr	Branch if not Negative	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-45

Legend: ⇳ set or cleared; ⇩ *may* be cleared, but never set; ⇧ *may* be set, but never cleared; '1' always set; '0' always cleared; — unchanged

Note: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.

Table 6-3: dsPIC30F Instruction Set Summary Table (Continued)

Assembly Syntax Mnemonic,Operands		Description	Words	Cycles	OA	OB	SA	SB	OAB	SAB	DC	N	OV	Z	C	Page #
BRA	NOV,Expr	Branch if not Overflow	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-46
BRA	NZ,Expr	Branch if not Zero	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-47
BRA	OA,Expr	Branch if Accumulator A overflow	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-48
BRA	OB,Expr	Branch if Accumulator B overflow	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-49
BRA	OV,Expr	Branch if Overflow	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-50
BRA	SA,Expr	Branch if Accumulator A saturated	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-51
BRA	SB,Expr	Branch if Accumulator B saturated	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-52
BRA	Z,Expr	Branch if Zero	1	1 (2)	—	—	—	—	—	—	—	—	—	—	—	5-53
BSET	f,#bit4	Bit set f	1	1	—	—	—	—	—	—	—	—	—	—	—	5-54
BSET	Ws,#bit4	Bit set Ws	1	1	—	—	—	—	—	—	—	—	—	—	—	5-55
BSW.C	Ws,Wb	Write C bit to Ws<Wb>	1	1	—	—	—	—	—	—	—	—	—	—	—	5-56
BSW.Z	Ws,Wb	Write Z bit to Ws<Wb>	1	1	—	—	—	—	—	—	—	—	—	—	—	5-56
BTG	f,#bit4	Bit toggle f	1	1	—	—	—	—	—	—	—	—	—	—	—	5-58
BTG	Ws,#bit4	Bit toggle Ws	1	1	—	—	—	—	—	—	—	—	—	—	—	5-59
BTSC	f,#bit4	Bit test f, skip if clear	1	1 (2 or 3)	—	—	—	—	—	—	—	—	—	—	—	5-60
BTSC	Ws,#bit4	Bit test Ws, skip if clear	1	1 (2 or 3)	—	—	—	—	—	—	—	—	—	—	—	5-62
BTSS	f,#bit4	Bit test f, skip if set	1	1 (2 or 3)	—	—	—	—	—	—	—	—	—	—	—	5-64
BTSS	Ws,#bit4	Bit test Ws, skip if set	1	1 (2 or 3)	—	—	—	—	—	—	—	—	—	—	—	5-65
BTST	f,#bit4	Bit test f	1	1	—	—	—	—	—	—	—	—	—	⇕	—	5-67
BTST.C	Ws,#bit4	Bit test Ws to C	1	1	—	—	—	—	—	—	—	—	—	—	⇕	5-68
BTST.Z	Ws,#bit4	Bit test Ws to Z	1	1	—	—	—	—	—	—	—	—	—	⇕	—	5-68
BTST.C	Ws,Wb	Bit test Ws<Wb> to C	1	1	—	—	—	—	—	—	—	—	—	—	⇕	5-69
BTST.Z	Ws,Wb	Bit test Ws<Wb> to Z	1	1	—	—	—	—	—	—	—	—	—	⇕	—	5-69
BTSTS	f,#bit4	Bit test then set f	1	1	—	—	—	—	—	—	—	—	—	⇕	—	5-71
BTSTS.C	Ws,#bit4	Bit test Ws to C then set	1	1	—	—	—	—	—	—	—	—	—	—	⇕	5-72
BTSTS.Z	Ws,#bit4	Bit test Ws to Z then set	1	1	—	—	—	—	—	—	—	—	—	⇕	—	5-72
CALL	Expr	Call subroutine	2	2	—	—	—	—	—	—	—	—	—	—	—	5-73
CALL	Wn	Call indirect subroutine	1	2	—	—	—	—	—	—	—	—	—	—	—	5-74
CLR	f	f = 0x0000	1	1	—	—	—	—	—	—	—	—	—	—	—	5-75
CLR	WREG	WREG = 0x0000	1	1	—	—	—	—	—	—	—	—	—	—	—	5-75
CLR	Wd	Wd = 0	1	1	—	—	—	—	—	—	—	—	—	—	—	5-76
CLR	Acc,Wx,Wxd,Wy,Wyd,AWB	Clear Accumulator	1	1	0	0	0	0	0	0	—	—	—	—	—	5-77
CLRWDT		Clear Watchdog Timer	1	1	—	—	—	—	—	—	—	—	—	—	—	5-79
COM	f {,WREG}	Destination = $\bar{f}$	1	1	—	—	—	—	—	—	—	⇕	—	⇕	—	5-80
COM	Ws,Wd	Wd = $\overline{Ws}$	1	1	—	—	—	—	—	—	—	⇕	—	⇕	—	5-81

Legend: ⇕ set or cleared; ⇩ *may* be cleared, but never set; ⇧ *may* be set, but never cleared; '1' always set; '0' always cleared; — unchanged

Note: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.

Table 6-3: dsPIC30F Instruction Set Summary Table (Continued)

Assembly Syntax Mnemonic,Operands		Description	Words	Cycles	OA	OB	SA	SB	OAB	SAB	DC	N	OV	Z	C	Page #
CP	f	Compare (f – WREG)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-82
CP	Wb,#lit5	Compare (Wb – lit5)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-83
CP	Wb,Ws	Compare (Wb – Ws)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-84
CP0	f	Compare (f – 0x0000)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-85
CP0	Ws	Compare (Ws – 0x0000)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-86
CPB	f	Compare with borrow (f – WREG – $\overline{C}$)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇩	⇳	5-87
CPB	Wb,#lit5	Compare with borrow (Wb – lit5 – $\overline{C}$)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇩	⇳	5-88
CPB	Wb,Ws	Compare with borrow (Wb – Ws – $\overline{C}$)	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇩	⇳	5-89
CPSEQ	Wb, Wn	Compare (Wb with Wn), skip if =	1	1 (2 or 3)	—	—	—	—	—	—	—	—	—	—	—	5-91
CPSGT	Wb, Wn	Signed Compare (Wb with Wn), skip if >	1	1 (2 or 3)	—	—	—	—	—	—	—	—	—	—	—	5-92
CPSLT	Wb, Wn	Signed Compare (Wb with Wn), skip if <	1	1 (2 or 3)	—	—	—	—	—	—	—	—	—	—	—	5-93
CPSNE	Wb, Wn	Signed Compare (Wb with Wn), skip if ≠	1	1 (2 or 3)	—	—	—	—	—	—	—	—	—	—	—	5-94
DAW.B	Wn	Wn = decimal adjust Wn	1	1	—	—	—	—	—	—	—	—	—	—	⇳	5-95
DEC	f {,WREG}	Destination = f – 1	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-96
DEC	Ws,Wd	Wd = Ws – 1	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-97
DEC2	f {,WREG}	Destination = f – 2	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-98
DEC2	Ws,Wd	Wd = Ws – 2	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-99
DISI	#lit14	Disable interrupts for lit14 instruction cycles	1	1	—	—	—	—	—	—	—	—	—	—	—	5-100
DIV.S	Wm, Wn	Signed 16/16-bit integer divide	1	18	—	—	—	—	—	—	—	⇳	⇳	⇳	⇳	5-101
DIV.SD	Wm, Wn	Signed 32/16-bit integer divide	1	18	—	—	—	—	—	—	—	⇳	⇳	⇳	⇳	5-101
DIV.U	Wm, Wn	Unsigned 16/16-bit integer divide	1	18	—	—	—	—	—	—	—	0	⇳	⇳	⇳	5-103
DIV.UD	Wm, Wn	Unsigned 32/16-bit integer divide	1	18	—	—	—	—	—	—	—	0	⇳	⇳	⇳	5-103
DIVF	Wm, Wn	Signed 16/16-bit fractional divide	1	18	—	—	—	—	—	—	—	⇳	⇳	⇳	⇳	5-105
DO	#lit14, Expr	Do code to PC+Expr, (lit14+1) times	2	2	—	—	—	—	—	—	—	—	—	—	—	5-107
DO	Wn, Expr	Do code to PC+Expr, (Wn+1) times	2	2	—	—	—	—	—	—	—	—	—	—	—	5-109
ED	Wm*Wm,Acc,Wx,Wy,Wxd	Euclidean distance (no accumulate)	1	1	0	0	—	—	0	—	—	—	—	—	—	5-111
EDAC	Wm*Wm,Acc,Wx,Wy,Wxd	Euclidean distance	1	1	⇳	⇳	⇧	⇧	⇳	⇧	—	—	—	—	—	5-113
EXCH	Wns,Wnd	Swap Wns and Wnd	1	1	—	—	—	—	—	—	—	—	—	—	—	5-115
FBCL	Ws,Wnd	Find bit change from left (MSb) side	1	1	—	—	—	—	—	—	—	—	—	—	⇳	5-116
FF1L	Ws,Wnd	Find first one from left (MSb) side	1	1	—	—	—	—	—	—	—	—	—	—	⇳	5-118
FF1R	Ws,Wnd	Find first one from right (LSb) side	1	1	—	—	—	—	—	—	—	—	—	—	⇳	5-120
GOTO	Expr	Go to address	2	2	—	—	—	—	—	—	—	—	—	—	—	5-122
GOTO	Wn	Go to address indirectly	1	2	—	—	—	—	—	—	—	—	—	—	—	5-123

Legend: ⇳ set or cleared; ⇩ *may* be cleared, but never set; ⇧ *may* be set, but never cleared; ‘1’ always set; ‘0’ always cleared; — unchanged

Note: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.

Table 6-3: dsPIC30F Instruction Set Summary Table (Continued)

Assembly Syntax Mnemonic	Operands	Description	Words	Cycles	OA	OB	SA	SB	OAB	SAB	DC	N	OV	Z	C	Page #
INC	f {,WREG}	Destination = f + 1	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-124
INC	Ws,Wd	Wd = Ws + 1	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-125
INC2	f {,WREG}	Destination = f + 2	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-126
INC2	Ws,Wd	Wd = Ws + 2	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-127
IOR	f {,WREG}	Destination = f .IOR. WREG	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-128
IOR	#lit10,Wn	Wn = lit10 .IOR. Wn	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-129
IOR	Wb,#lit5,Wd	Wd = Wb .IOR. lit5	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-130
IOR	Wb,Ws,Wd	Wd = Wb .IOR. Ws	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-131
LAC	Ws,#Slit4, Acc	Load Accumulator	1	1	⇳	⇳	⇧	⇧	⇳	⇧	—	—	—	—	—	5-133
LNK	#lit14	Link frame pointer	1	1	—	—	—	—	—	—	—	—	—	—	—	5-135
LSR	f {,WREG}	Destination = logical right shift f	1	1	—	—	—	—	—	—	—	⇩	—	⇳	⇳	5-136
LSR	Ws,Wd	Wd = logical right shift Ws	1	1	—	—	—	—	—	—	—	⇩	—	⇳	⇳	5-137
LSR	Wb,#lit4,Wnd	Wnd = logical right shift Wb by lit4	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-139
LSR	Wb,Wns,Wnd	Wnd = logical right shift Wb by Wns	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-140
MAC	Wm*Wn,Acc,Wx,Wxd,Wy,Wyd,AWB	Multiply and accumulate	1	1	⇳	⇳	⇧	⇧	⇳	⇧	—	—	—	—	—	5-141
MAC	Wm*Wm,Acc,Wx,Wxd,Wy,Wyd,	Square and accumulate	1	1	⇳	⇳	⇧	⇧	⇳	⇧	—	—	—	—	—	5-143
MOV	f {,WREG}	Move f to destination	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-145
MOV	WREG,f	Move WREG to f	1	1	—	—	—	—	—	—	—	—	—	—	—	5-146
MOV	f,Wnd	Move f to Wnd	1	1	—	—	—	—	—	—	—	—	—	—	—	5-147
MOV	Wns,f	Move Wns to f	1	1	—	—	—	—	—	—	—	—	—	—	—	5-148
MOV.B	#lit8,Wnd	Move 8-bit unsigned literal to Wnd	1	1	—	—	—	—	—	—	—	—	—	—	—	5-149
MOV	#lit16,Wnd	Move 16-bit literal to Wnd	1	1	—	—	—	—	—	—	—	—	—	—	—	5-150
MOV	[Wns+Slit10],Wnd	Move [Wns + Slit10] to Wnd	1	1	—	—	—	—	—	—	—	—	—	—	—	5-151
MOV	Wns,[Wnd+Slit10]	Move Wns to [Wnd + Slit10]	1	1	—	—	—	—	—	—	—	—	—	—	—	5-152
MOV	Ws,Wd	Move Ws to Wd	1	1	—	—	—	—	—	—	—	—	—	—	—	5-153
MOV.D	Ws,Wnd	Move double Ws to Wnd:Wnd+1	1	2	—	—	—	—	—	—	—	—	—	—	—	5-155
MOV.D	Wns,Wd	Move double Wns:Wns+1 to Wd	1	2	—	—	—	—	—	—	—	—	—	—	—	5-157
MOVSAC	Acc,Wx,Wxd,Wy,Wyd,AWB	Move [Wx] to Wxd, and [Wy] to Wyd	1	1	—	—	—	—	—	—	—	—	—	—	—	5-159
MPY	Wm*Wn,Acc,Wx,Wxd,Wy,Wyd	Multiply Wn by Wm to accumulator	1	1	0	0	—	—	0	—	—	—	—	—	—	5-161
MPY	Wm*Wm,Acc,Wx,Wxd,Wy,Wyd	Square to Accumulator	1	1	0	0	—	—	0	—	—	—	—	—	—	5-163
MPY.N	Wm*Wn,Acc,Wx,Wxd,Wy,Wyd	-(Multiply Wn by Wm) to Accumulator	1	1	0	0	—	—	0	—	—	—	—	—	—	5-165
MSC	Wm*Wn,Acc,Wx,Wxd,Wy,Wyd,AWB	Multiply and subtract from Accumulator	1	1	⇳	⇳	⇧	⇧	⇳	⇧	—	—	—	—	—	5-167
MUL	f	W3:W2 = f * WREG	1	1	—	—	—	—	—	—	—	—	—	—	—	5-169
MUL.SS	Wb,Ws,Wnd	{Wnd+1,Wnd} = sign(Wb) * sign(Ws)	1	1	—	—	—	—	—	—	—	—	—	—	—	5-170

Legend: ⇳ set or cleared; ⇩ *may* be cleared, but never set; ⇧ *may* be set, but never cleared; ‘1’ always set; ‘0’ always cleared; — unchanged

Note: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.

Table 6-3: dsPIC30F Instruction Set Summary Table (Continued)

Assembly Syntax Mnemonic,Operands		Description	Words	Cycles	OA	OB	SA	SB	OAB	SAB	DC	N	OV	Z	C	Page #
MUL.SU	Wb,#lit5,Wnd	{Wnd+1,Wnd} = sign(Wb) * unsign(lit5)	1	1	—	—	—	—	—	—	—	—	—	—	—	5-172
MUL.SU	Wb,Ws,Wnd	{Wnd+1,Wnd} = sign(Wb) * unsign(Ws)	1	1	—	—	—	—	—	—	—	—	—	—	—	5-174
MUL.US	Wb,Ws,Wnd	{Wnd+1,Wnd} = unsign(Wb) * sign(Ws)	1	1	—	—	—	—	—	—	—	—	—	—	—	5-176
MUL.UU	Wb,#lit5,Wnd	{Wnd+1,Wnd} = unsign(Wb) * unsign(lit5)	1	1	—	—	—	—	—	—	—	—	—	—	—	5-178
MUL.UU	Wb,Ws,Wnd	{Wnd+1,Wnd} = unsign(Wb) * unsign(Ws)	1	1	—	—	—	—	—	—	—	—	—	—	—	5-179
NEG	f {,WREG}	Destination = $\bar{f}$ + 1	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-181
NEG	Ws,Wd	Wd = $\overline{Ws}$ + 1	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-182
NEG	Acc	Negate Accumulator	1	1	⇳	⇳	⇧	⇧	⇳	⇧	—	—	—	—	—	5-183
NOP		No operation	1	1	—	—	—	—	—	—	—	—	—	—	—	5-184
NOPR		No operation	1	1	—	—	—	—	—	—	—	—	—	—	—	5-185
POP	f	Pop TOS to f	1	1	—	—	—	—	—	—	—	—	—	—	—	5-186
POP	Wd	Pop TOS to Wd	1	1	—	—	—	—	—	—	—	—	—	—	—	5-187
POP.D	Wnd	Pop double from TOS to Wnd:Wnd+1	1	2	—	—	—	—	—	—	—	—	—	—	—	5-188
POP.S		Pop shadow registers	1	1	—	—	—	—	—	—	⇳	⇳	⇳	⇳	⇳	5-189
PUSH	f	Push f to TOS	1	1	—	—	—	—	—	—	—	—	—	—	—	5-190
PUSH	Ws	Push Ws to TOS	1	1	—	—	—	—	—	—	—	—	—	—	—	5-191
PUSH.D	Wns	Push double Wns:Wns+1 to TOS	1	2	—	—	—	—	—	—	—	—	—	—	—	5-192
PUSH.S		Push shadow registers	1	1	—	—	—	—	—	—	—	—	—	—	—	5-193
PWRSAV	#lit1	Enter Power Saving mode	1	1	—	—	—	—	—	—	—	—	—	—	—	5-194
RCALL	Expr	Relative call	1	2	—	—	—	—	—	—	—	—	—	—	—	5-195
RCALL	Wn	Computed call	1	2	—	—	—	—	—	—	—	—	—	—	—	5-196
REPEAT	#lit14	Repeat next instruction (lit14+1) times	1	1	—	—	—	—	—	—	—	—	—	—	—	5-197
REPEAT	Wn	Repeat next instruction (Wn+1) times	1	1	—	—	—	—	—	—	—	—	—	—	—	5-198
RESET		Software device RESET	1	1	—	—	—	—	—	—	—	—	—	—	—	5-200
RETFIE		Return from interrupt enable	1	3 (2)	—	—	—	—	—	—	—	⇳	⇳	⇳	⇳	5-201
RETLW	#lit10,Wn	Return with lit10 in Wn	1	3 (2)	—	—	—	—	—	—	—	—	—	—	—	5-202
RETURN		Return from subroutine	1	3 (2)	—	—	—	—	—	—	—	—	—	—	—	5-203
RLC	f {,WREG}	Destination = rotate left through Carry f	1	1	—	—	—	—	—	—	—	⇳	—	⇳	⇳	5-204
RLC	Ws,Wd	Wd = rotate left through Carry Ws	1	1	—	—	—	—	—	—	—	⇳	—	⇳	⇳	5-205
RLNC	f {,WREG}	Destination = rotate left (no Carry) f	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-207
RLNC	Ws,Wd	Wd = rotate left (no Carry) Ws	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-208
RRC	f {,WREG}	Destination = rotate right through Carry f	1	1	—	—	—	—	—	—	—	⇳	—	⇳	⇳	5-210
RRC	Ws,Wd	Wd = rotate right through Carry Ws	1	1	—	—	—	—	—	—	—	⇳	—	⇳	⇳	5-211
RRNC	f {,WREG}	Destination = rotate right (no Carry) f	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-213
RRNC	Ws,Wd	Wd = rotate right (no Carry) Ws	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-214

Legend: ⇳ set or cleared; ⇩ *may* be cleared, but never set; ⇧ *may* be set, but never cleared; '1' always set; '0' always cleared; — unchanged

Note: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.

Table 6-3: dsPIC30F Instruction Set Summary Table (Continued)

Mnemonic	Assembly Syntax Mnemonic,Operands	Description	Words	Cycles	OA	OB	SA	SB	OAB	SAB	DC	N	OV	Z	C	Page #
SAC	Acc,#Slit4,Wd	Store Accumulator	1	1	—	—	—	—	—	—	—	—	—	—	—	5-216
SAC.R	Acc,#Slit4,Wd	Store rounded Accumulator	1	1	—	—	—	—	—	—	—	—	—	—	—	5-218
SE	Ws,Wd	Wd = sign-extended Ws	1	1	—	—	—	—	—	—	—	⇕	—	⇕	⇕	5-220
SETM	f	f = `0xFFFF`	1	1	—	—	—	—	—	—	—	—	—	—	—	5-221
SETM	WREG	WREG = `0xFFFF`	1	1	—	—	—	—	—	—	—	—	—	—	—	5-221
SETM	Ws	Ws = `0xFFFF`	1	1	—	—	—	—	—	—	—	—	—	—	—	5-222
SFTAC	Acc,#Slit6	Arithmetic shift accumulator by Slit6	1	1	⇕	⇕	⇧	⇧	⇕	⇧	—	—	—	—	—	5-223
SFTAC	Acc,Wn	Arithmetic shift accumulator by (Wn)	1	1	⇕	⇕	⇧	⇧	⇕	⇧	—	—	—	—	—	5-224
SL	f {,WREG}	Destination = arithmetic left shift f	1	1	—	—	—	—	—	—	—	⇕	—	⇕	⇕	5-225
SL	Ws,Wd	Wd = arithmetic left shift Ws	1	1	—	—	—	—	—	—	—	⇕	—	⇕	⇕	5-226
SL	Wb,#lit4,Wnd	Wnd = left shift Wb by lit4	1	1	—	—	—	—	—	—	—	⇕	—	⇕	—	5-228
SL	Wb,Wns,Wnd	Wnd = left shift Wb by Wns	1	1	—	—	—	—	—	—	—	⇕	—	⇕	—	5-229
SUB	f {,WREG}	Destination = f – WREG	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇕	⇕	5-230
SUB	#lit10,Wn	Wn = Wn – lit10	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇕	⇕	5-231
SUB	Wb,#lit5,Wd	Wd = Wb – lit5	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇕	⇕	5-232
SUB	Wb,Ws,Wd	Wd = Wb – Ws	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇕	⇕	5-233
SUB	Acc	Subtract Accumulators	1	1	⇕	⇕	⇧	⇧	⇕	⇧	—	—	—	—	—	5-235
SUBB	f {,WREG}	destination = f – WREG – ($\overline{C}$)	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇩	⇕	5-236
SUBB	#lit10,Wn	Wn = Wn – lit10 – ($\overline{C}$)	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇩	⇕	5-237
SUBB	Wb,#lit5,Wd	Wd = Wb – lit5 – ($\overline{C}$)	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇩	⇕	5-238
SUBB	Wb,Ws,Wd	Wd = Wb – Ws – ($\overline{C}$)	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇩	⇕	5-239
SUBBR	f {,WREG}	Destination = WREG – f – ($\overline{C}$)	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇩	⇕	5-241
SUBBR	Wb,#lit5,Wd	Wd = lit5 – Wb – ($\overline{C}$)	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇩	⇕	5-242
SUBBR	Wb,Ws,Wd	Wd = Ws – Wb – ($\overline{C}$)	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇩	⇕	5-243
SUBR	f {,WREG}	Destination = WREG – f	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇕	⇕	5-245
SUBR	Wb,#lit5,Wd	Wd = lit5 – Wb	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇕	⇕	5-246
SUBR	Wb,Ws,Wd	Wd = Ws – Wb	1	1	—	—	—	—	—	—	⇕	⇕	⇕	⇕	⇕	5-247
SWAP	Wn	Wn = byte or nibble swap Wn	1	1	—	—	—	—	—	—	—	—	—	—	—	5-249
TBLRDH	Ws,Wd	Read high program word to Wd	1	2	—	—	—	—	—	—	—	—	—	—	—	5-250
TBLRDL	Ws,Wd	Read low program word to Wd	1	2	—	—	—	—	—	—	—	—	—	—	—	5-252
TBLWTH	Ws,Wd	Write Ws to high program word	1	2	—	—	—	—	—	—	—	—	—	—	—	5-254
TBLWTL	Ws,Wd	Write Ws to low program word	1	2	—	—	—	—	—	—	—	—	—	—	—	5-256
ULNK		Unlink frame pointer	1	1	—	—	—	—	—	—	—	—	—	—	—	5-258

Legend: ⇕ set or cleared; ⇩ *may* be cleared, but never set; ⇧ *may* be set, but never cleared; '`1`' always set; '`0`' always cleared; — unchanged

Note: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.

Table 6-3: dsPIC30F Instruction Set Summary Table (Continued)

Assembly Syntax Mnemonic	Operands	Description	Words	Cycles	OA	OB	SA	SB	OAB	SAB	DC	N	OV	Z	C	Page #
XOR	f {,WREG}	Destination = f .XOR. WREG	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-259
XOR	#lit10,Wn	Wn = lit10 .XOR. Wn	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-260
XOR	Wb,#lit5,Wd	Wd = Wb .XOR. lit5	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-261
XOR	Wb,Ws,Wd	Wd = Wb .XOR. Ws	1	1	—	—	—	—	—	—	—	⇳	—	⇳	—	5-262
ZE	Ws,Wd	Wd = zero-extended Ws	1	1	—	—	—	—	—	—	—	0	—	⇳	1	5-264

Legend: ⇳ set or cleared; ⇩ *may* be cleared, but never set; ⇧ *may* be set, but never cleared; '1' always set; '0' always cleared; — unchanged

Note: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.

INDEX

 Advance Information

M

N

O

P

NOTES:

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com

Atlanta
3780 Mansell Road, Suite 130
Alpharetta, GA 30022
Tel: 770-640-0034 Fax: 770-640-0307

Boston
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

Chicago
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423 Fax: 972-818-2924

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Kokomo
2767 S. Albright Road
Kokomo, IN 46902
Tel: 765-864-8360 Fax: 765-864-8387

Los Angeles
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

Phoenix
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-4338

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
6285 Northam Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

ASIA/PACIFIC

Australia
Microchip Technology Australia Pty Ltd
Marketing Support Division
Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733 Fax: 61-2-9868-6755

China - Beijing
Microchip Technology Consulting (Shanghai) Co., Ltd., Beijing Liaison Office
Unit 915
Bei Hai Wan Tai Bldg.
No. 6 Chaoyangmen Beidajie
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Chengdu
Microchip Technology Consulting (Shanghai) Co., Ltd., Chengdu Liaison Office
Rm. 2401-2402, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-86766200 Fax: 86-28-86766599

China - Fuzhou
Microchip Technology Consulting (Shanghai) Co., Ltd., Fuzhou Liaison Office
Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506 Fax: 86-591-7503521

China - Hong Kong SAR
Microchip Technology Hongkong Ltd.
Unit 901-6, Tower 2, Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200 Fax: 852-2401-3431

China - Shanghai
Microchip Technology Consulting (Shanghai) Co., Ltd.
Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel: 86-21-6275-5700 Fax: 86-21-6275-5060

China - Shenzhen
Microchip Technology Consulting (Shanghai) Co., Ltd., Shenzhen Liaison Office
Rm. 1812, 18/F, Building A, United Plaza
No. 5022 Binhe Road, Futian District
Shenzhen 518033, China
Tel: 86-755-82901380 Fax: 86-755-82966626

China - Qingdao
Rm. B505A, Fullhope Plaza,
No. 12 Hong Kong Central Rd.
Qingdao 266071, China
Tel: 86-532-5027355 Fax: 86-532-5027205

India
Microchip Technology Inc.
India Liaison Office
Marketing Support Division
Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaugnessey Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax: 91-80-2290062

Japan
Microchip Technology Japan K.K.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471- 6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel: 65-6334-8870 Fax: 65-6334-8850

Taiwan
Microchip Technology (Barbados) Inc.,
Taiwan Branch
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Austria
Microchip Technology Austria GmbH
Durisolstrasse 2
A-4600 Wels
Austria
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

Denmark
Microchip Technology Nordic ApS
Regus Business Centre
Lautrup hoj 1-3
Ballerup DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France
Microchip Technology SARL
Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Microchip Technology GmbH
Steinheilstrasse 10
D-85737 Ismaning, Germany
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy
Microchip Technology SRL
Via Quasimodo, 12
20025 Legnano (MI)
Milan, Italy
Tel: 39-0331-742611 Fax: 39-0331-466781

United Kingdom
Microchip Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44-118 921-5820

03/25/03

